DISS. ETH NO. 20126

Automatic Verification of Heap Structures with Stereotypes

A dissertation submitted to
ETH Zurich

for the degree of
Doctor of Sciences

presented by
Arsenii Rudich
Master of Computer Science

born May 14, 1982
citizen of Ukraine

accepted on the recommendation of

Prof. Dr. Peter Müller, examiner
Prof. Dr. David Basin, co-examiner
Prof. Dr. Anindya Banerjee, co-examiner

2011

Abstract

This thesis is dedicated to the automatic and formal verification of the heap properties of object oriented programs. Program verification is the check that a given program satisfies given properties. Program verification is called formal if both the semantics of the specifications and the program execution are defined formally as mathematical entities. The verification is called automatic if it is performed automatically without interaction or with limited interaction with a user.

Our approach is targeted towards the verification of the preservation of heap-topological properties. It is also aimed towards the verification of the effects and the frame properties of the program statements.

Automatic verification of heap structures is crucial for the verification of multi-object invariants, the verification of concurrent programs (e.g., absence of race conditions and deadlocks), software engineering (e.g., enabling encapsulation and modular development, handling design patterns), and the verification of security properties (e.g., isolation).

We present a novel approach to the verification of heap structures. Our approach is based on the notion of *stereotypes*. The aim behind a stereotype is to collect in one entity all reusable specifications which are relevant to the heap structure. Stereotypes provide, amongst other information, approximations of the transitive closures of relevant fields; information which is not generally available in a first-order logic. These approximations enable the verification of heap properties in automatic fist order logic theorem provers. The usage of our stereotype based approach has advantages such as reduction of the specification overhead, prevention of specification duplication, the prevention of proof duplication, and the improvement of the readability of the specifications.

To evaluate the stereotype based approach we have specified and verified several design patterns and data structures. All examples are verified in the verification language Boogie. One of the verified examples is the Priority Inheritance Protocol. According to our knowledge it is the first automatic verification of the Priority Inheritance Protocol.

Zusammenfassung

Diese Doktorarbeit ist der automatischen formalen Verifikation von Heap-Eigenschaften objektorientierter Programme gewidmet. Programm-Verifikation ist die Überprüfung, dass ein bestimmtes Programm gegebenen Eigenschaften erfüllt. Programm-Verifikation ist formal, wenn die Semantik der Spezifikation wie auch der Programm-Ausführung mathematisch definiert ist. Die Überprüfung wird als automatisch bezeichnet, wenn sie ohne oder mit begrenzter Interaktion durch einem Benutzer durchgeführt wird.

Unser Ansatz ist auf die Überprüfung der Erhaltung topologischer Heap-Eigenschaften ausgerichtet. Auch die Prüfung von Frame-Conditions ist Ziel dieser Arbeit.

Automatische Überprüfung von Heap-Strukturen ist entscheidend für die Überprüfung eines Verbundes von Objekt-Invarianten, die Überprüfung nebenläufiger Programme (z.B. Fehlen von Race-Conditions und Deadlocks), Software-Engineering Aspekte (z. B. Kapselung, Modularität, Handhabung von Design Patterns), und die Überprüfung Sicherheit-Eigenschaften (z. B. Isolation).

Wir präsentieren einen neuartigen Ansatz zur Überprüfung von Heap-Strukturen. Unser Ansatz basiert auf der Idee von Stereotypen. Das Ziel eines Stereotyps ist, alle wiederverwendbaren Spezifikationen, für die Heap-Struktur zusammenzufassen. Dazu gehören unter anderem die Stereotypen-Annäherungen an die transitive Hülle von entsprechenden Felder in First-Order Logik. Diese Näherungen ermöglichen die Überprüfung von Heap-Eigenschaften mittels automatischer First-Order Logik Theorembeweisern. Die Nutzung unseres Stereotyp Ansatzes bietet Vorteile wie Reduzierung des Spezifikations-Overhead, Vermeidung von doppelter Spezifikationen, Verhinderung von Beweis Vervielfältigung und Verbesserung der Lesbarkeit der Spezifikationen.

Zur Beurteilung der Stereotyp Ansatz haben wir mehrere Design-Patterns und Datenstrukturen spezifiziert und verifiziert. Alle Beispiele sind in der Verifikationssprache Boogie überprüft. Eines der überprüften Beispiele ist das Priority Inheritance Protocol. Nach unserer Kenntnis handelt es sich um die erste automatische Überprüfung dieses Protokolls.

Acknowledgments

First of all I would like to sincerely thank my professor Peter Müller for his invaluable contribution, guidance, and attentive supervision. I would like to specially thank him for the patience, for his support in seeing this thesis successfully completed, as well as for all the great scientific discussions which we have had (it was a lot of fun and one of the best parts of the thesis).

I would like to thank my reviewers Dr. David Basin and Dr. Anindya Banerjee for their truly heroic efforts in reading this thesis. The provided feedback was instrumental in the formation of this thesis.

I am very grateful to Dr. Alex Summers and Dr. Sophia Drossopoulou for the extremely useful meeting which we had in Imperial College in 2009. During that time we had multiple discussions, which were crucial for crystallization of the ideas presented in this thesis.

After Alex moved to ETH Zurich from Imperial College he continued to provide me with the extremely useful comments regarding the theory described in this thesis. I also would like to thank Alex for his linguistic comments.

Dr. Hermann Lehner was my ETH officemate for the past 6 years. I truly believe that it was one of my biggest luck in ETH to get such a fantastic officemate. Herman constantly helped me during my work in ETH starting from meeting me at the airport during my first visit to ETH, finding my first apartment in Zurich, and finishing with the translation of my thesis' abstract to German. Hermann gave me a great deal of moral support, and needless to say how many exciting scientific discussions we have had all these years. Not only he is my ex- officemate but also a great friend.

I would also like to thank Dr. Adam Darvas, whom I had a very productive scientific collaboration with. Much gratitude also goes to my colleagues Joseph Ruskiewicz, Dr. Laura Kovac, Dr. Werner Dietl, Cédric Favre, Valentin Wüstholz, Uri Juhasz, Milos Novacek, Dr. Ioannis Kassios, Dr. Pietro Ferrara from the Programming Methodology Group; with them I shared teaching obligations, scientific discussions and lunches.

Marlies Weissert is like a mother for the PM group. She takes care of all these scientific nerds, solves the bureaucratic stuff, helps with German translation and gives us both moral and administrative support. Many thanks to her.

I would like to thank students Benjamin Lutz, Annetta Schaad, Mathias Ottiger, Yoshimi Takano, Christoph Studer whom I had the honor to supervise during my work in ETH. Yoshimi Takano is the coolest master student ever.

There many people whom I am grateful to for the moral support and useful advice of various nature. I do not have enough space here to mention all of them, but nevertheless thanks a lot to all of you buddies. I would like to especially thank Alexander Vezhnevets, with whom I shared an apartment for several years and had a lot of exiting discussions. He considerably affected my perception of science and life. Also a special thank goes to Roman Mitin with whom I had inspiring discourses about the perspectives of programming languages and shared some extra curriculum activities (e.g. probability theory learning). I am very grateful to Jevgeny Chinkarev for wonderful conversations regarding philosophy, religion and life, and many of good moments we shared together. It would be unfair not to mention Schostka's buddies whom it was always nice to reset my brain from time to time with.

I would like to show much gratitude to my family; to my father Rudich Ievgen, to my mother Korotchenko Tetiana, to my sister Ruslana Yanovich and my niece Arina Yanovich. Thank you very much my loved ones for all the moral support and encouragement you gave me to finish my Ph.D., for being with me in good and bad times and for your visionary advices.

This chapter would not be complete without saying thank you to my beautiful wife Margarita Timofeeva, the one that brings meaning into my life, the one that inspires me every day, the one that I am going to share the rest of my life with.

Contents

1 Introduction **1**

1.1 Applications of the verification of heap properties 4

1.1.1 Verification of multiple objects invariants 4

1.1.2 Verification of concurrent programs 4

1.1.3 Software engineering 5

1.1.4 Verification of security properties 5

1.2 Existing approaches for the verification of heap properties 6

1.3 Approaches based on user-provided updates of a transitive closure 8

1.4 Stereotypes 12

1.5 The project context 16

1.6 Outline 17

2 Motivating example: priority inheritance protocol **19**

3 Stereotype-based verification methodology **29**

3.1 Motivating example 30

3.2 Stereotypes 33

3.2.1 Stereotype items 34

3.2.2 Stereotype invariants 38

3.2.3 Stereotypes 44

3.3 Stereotype operations 45

3.3.1 Basic stereotype operations 47

3.3.2 Specification language of specifications 54

3.4 Usage of stereotypes for source code specification 57

3.4.1 Participation of a class in a stereotype slice 68

3.4.2 Ghost updates 70

3.4.3 Glue invariants 77

3.4.4 Stereotype operations for source code specification . . 81

3.4.5 Behavioral invariants 83

3.4.6 Summary 85

4 Stereotype examples **87**
4.1 `Sequence` . . . 87
4.2 `Tree` . . . 98

5 Examples of stereotype-based verification **115**
5.1 List . . . 115
5.2 Composite . . . 120
5.3 Disjoint-set forests . . . 125
5.4 PIP . . . 128
5.5 **Boogie** experiments . . . 136
5.5.1 Composite design pattern . . . 138
5.5.2 **PIP** . . . 141
5.5.3 List with iterator . . . 144
5.5.4 Universal transformation of the sequence stereotype . 146

6 Stereotype operations **149**
6.1 Methodology for set description . . . 151
6.2 Standard stereotype operation representation . . . 161
6.3 Specification language of specifications . . . 171
6.4 **SLS**-to-**SSOR** translation . . . 182
6.4.1 Basic operation call . . . 188
6.4.2 Composite operation call . . . 194
6.4.3 `skip` operator . . . 197
6.4.4 Conditional statement . . . 198
6.4.5 Sequential composition . . . 199
6.4.6 Parallel composition . . . 205
6.5 Universal transformation . . . 208

7 Related work **211**
7.1 Approaches based on decidable procedures . . . 212
7.1.1 Limits of decidable procedures . . . 213
7.1.2 Type systems for alias control . . . 215
7.1.3 Decidable heap logics . . . 216
7.1.4 Abstract interpretation of heap . . . 220
7.2 Approaches based on automatic theorem proving . . . 223
7.2.1 Automatic theorem provers for **FOL** . . . 225
7.2.2 Approximation of induction in **FOL** . . . 226
7.2.3 User provided updates of a transitive closure . . . 227
7.2.4 User provided inductive proofs . . . 230
7.3 Inductive theorem provers . . . 232
7.3.1 Explicit induction . . . 233
7.3.2 Proof by consistency . . . 235

7.4 Approaches based on interactive theorem proving . . . 238
7.5 Mixed approaches . . . 239

8 Conclusion **241**
8.1 Summary and contributions . . . 241
8.2 Directions for future work . . . 245
8.3 Applications and consequences . . . 250

A Universal transformations for sequence stereotype **257**
A.1 Operation `addSetSequenceRelation` . . . 257
A.2 Operation `removeSetSequenceRelation` . . . 261

B Universal transformations for the tree stereotype **265**
B.1 Level 1 . . . 265
B.1.1 Operation `addSetTreeRelation1` . . . 265
B.1.2 Operation `removeSetTreeRelation1` . . . 268
B.2 Level 2 . . . 270
B.2.1 Operation `addSetTreeRelation2` . . . 270
B.2.2 Operation `removeSetTreeRelation2` . . . 275
B.3 Level 3 . . . 278
B.3.1 Operation `addSetTreeRelation3` . . . 278
B.3.2 Operation `removeSetTreeRelation3` . . . 283

Chapter 1

Introduction

This thesis is dedicated to the **automatic** and **formal verification** of **heap properties of object oriented programs**. Let us explain this problem step by step.

Program verification is the check that a given program satisfies given properties. The program properties are also called specifications. If each execution of a program satisfies the given specifications it is called correct. A program is called totally correct if each of its executions terminates and satisfies the given specifications. A program is called partially correct if some of its executions may not terminate but all executions which terminate satisfy the specifications. In this thesis we consider only partial correctness and when we say that the program is correct we always mean partial correctness. The result of the program verification is the answer to the question whether the given program is correct or not with regards to the given specifications.

The program verification is called **formal** if both the semantics of specifications and the program execution are defined formally as mathematical entities [62]. The semantics of the program execution is usually formalized as an operational or a denotational semantics. An operational semantics consists of the formal definition of the program state and the description of how various program statements change the program state. A denotational semantics defines the semantics of a program by translating it to a formally defined function which maps the program input to the program output.

The specifications are usually formalized by means of various logics. A program specification is the logic expression which describes the program state in a given program point or establishes a relation between the program states in the various program points. For instance, a method pre-condition is a specification which describes the program state before the execution execution of a method. Another example of a program specification is a method post-condition which relates the program states before and after a method execution.

The formal semantics of the specifications and the program execution can be used to reduce program verification to theorem proving. Usually it is achieved by means of Hoare logic [64] or the weakest pre-condition calculus [43] (we denote it as **WPC**). Hoare logic is a logic in which deduction rules can be used to prove the validity of the following specifications. If the property P holds before the execution of a program statement S and the execution of S terminates, then the property Q holds after the execution of S. Such a specification of the program statement S is called Hoare triple and denoted as $\{P\}S\{Q\}$. The Hoare logic can be directly applied to verify a program annotated with certain specifications.

The weakest pre-condition calculus is a predicate transformation which for a given program statement S and post-condition Q produces the weakest pre-condition P such that $\{P\}S\{Q\}$ holds. The **WPC** can be used to produce from a program annotated with specifications a formula which holds if and only if the program correct. This formula is called proof obligation. Program verification can be reduced to checking the validity of the proof obligation.

The validity of a proof obligation can be checked with the help of an interactive [116, 113, 15] or an automatic [27, 37] theorem prover. In the first case a user provides a proof of the program correctness and an interactive theorem prover checks its validity. This verification is called interactive. On the other hand the proof obligation can be given to an automatic theorem prover. It tries to prove the proof obligation without interaction with the user. This verification called **automatic**. The obvious benefit of automatic verification is the essential reduction of the amount of work which has to be done by the user to verify the program. Nevertheless, automatic theorem plovers can proof validity of formulas of limited complexity. More complicated formulas cannot be verified automatically. The main challenge in automatic verification is to avoid formulas which cannot be handled by automatic theorem provers. As we mentioned above this thesis is dedicated to automatic verification.

Various verification techniques are targeted towards the verification of different properties. For instance, there are verification techniques which are targeted towards the verification of numerical properties [69] and information flow [106]. These thesis is dedicated to the program **verification of heap properties**.

In order to describe what heap properties are, first we have to decide how we formalize the program heap. The heap formalization includes a definition of the program heap state and the basic operations which are used for heap content access and modification.

One possible way to formalize the program heap is to define the program state as a map from one natural number to another number. The first natural number represents a memory cell address and the second natural number is the content of the memory cell. The basic heap operations are: the extraction of the value of a program cell and the setting of a new value of the program cell. The cell address is used in both operations to identify program cells. The model also supports address arithmetic. A cell address can be used in arithmetical expressions to compute a new cell address. These computed cell addresses can be used in heap operations.

The above model is used for the verification of the low level software, e.g. operation system or drivers verification. Nevertheless, it is a far too low level for our proposes. As we mentioned above, our methodology is targeted towards the **verification of object oriented programs** (denoted as **OOP**). In **OOP**-languages objects are treated as undividable entities. Therefore for **OOP**-languages we use the following heap model.

Each object is identified by its address. A special type reference contains object address values. A program heap is defined as a polymorphic map from a pair (object address, object field) to a value of the object field's type. A pair (object address, field) is called location. Obtaining a location value and setting of a new location value are the basic heap operations.

We can think of the heap state as a labeled graph which is constructed in the following way. The graph nodes are objects. There is a directed edge between objects o and o' labeled with the field name f if and only if $o.f = o'$. We call this graph heap shape or heap topology.

Our approach is targeted towards the verification of the following properties:

- *Preservation of heap topology properties by a program statement.* For instance, a well-formed linked list has to satisfy certain properties. E.g. if *next* is a field of list node which points to the next element of the list then the transitive closure of the relation induced by *next* has to be acyclic. This property has to be preserved by the heap manipulating program statements. For, instance a list reverse method has to preserve the list acyclicity property.

- *Specification of heap topology modifications by a program statement.* For instance, the list reverse method which we mentioned above has to specify how exactly it modifies the list. E.g. the method has to specify that the values of field *next* after the method execution are equal to the inversion of the relation induced by the field *next* before the method execution.

- *Frame properties of the program statements.* The frame properties specify which locations can be affected by a statement execution. Values of all other locations have to be preserved by the statement execution. For instance, the reverse method frame is the set of locations $\langle o, f\rangle$ where o is a list node of the affected list. Validity of the frame property guarantees that the only list which is affected by the method execution is the method receiver. Values of fields of all other objects are preserved by the method execution.

Let us now consider several applications of the verification of heap properties. The applications illustrate the practical significance of heap properties.

1.1 Applications of the verification of heap properties

Verification of the heap properties is crucial for various areas of computer science. Here is an incomplete list of areas which can benefit or are even hardly possible without a proper methodology for the specification of heap structures:

1.1.1 Verification of multiple objects invariants

A multiple objects invariant is a property which relates multiple objects. The number of objects which participate in a multiple objects invariant is potentially unbounded. On the other hand the same object can participate in an unbounded number of invariants. For these reasons verification of multiple objects invariants is challenging. To enable the verification of such invariants we have to be able to specify and verify on which objects an invariant depends and in which invariants an object participates. The heap topology can be used to extract this information.

1.1.2 Verification of concurrent programs

There are several properties of concurrent programs whose verification relies on properties of the heap topology:

- **Absence of race conditions.** A race condition arises when a thread writes a heap location and another thread reads or writes the same heap location in a concurrent way. The absence of the race conditions can be verified by checking the pairwise disjointness of the effects of the concurrently executed threads. The effect of a thread is defined by the frame of the thread body.

- **Deadlock freedom.** A deadlock arises when two or more threads lock the same objects in a different order. To prevent a deadlock it is enough to introduce a partial order over locked objects, and check that each thread locks them according to this order. Quite often such an order is retrieved from an acyclic data structure e.g. a list or a tree. Nevertheless, such an approach is sound only if the acyclicity of the heap structure is preserved during the program execution.

1.1.3 Software engineering

There following aspects of software engineering can be enabled by the verification of heap properties:

- **Encapsulation and modular development.** A strongly desired property for a software engineering project is modularity. It should be possible to split the whole project in independent modules in such a way that all communications between modules are going strictly via predefined module interfaces. The decomposition into independent modules enables independent module development and modules implementations interchangeability. Modularity also increases the project manageability. Nevertheless, modularity can be broken by an undesirable aliasing between modules. By an undesirable aliasing we imply the situation when there is an alias from a module A to a module B which is not mentioned in the interface of the module B but can be used by code of module A to change the state of module B. The heap topology specification can be used to prevent such undesirable aliasing between modules.

- **Design patterns.** A Design pattern is a reusable solution of a typical problem in software engineering. Since design patterns provide various benefits they are widely used in everyday software development. One of the aspects of design patterns is to control aliasing and semantical relations between the object instances which plays various roles in the design pattern. E.g. an observer observes a specific subject, an iterator iterates over a specific container, an adapter owns a specific adoptee. Nevertheless, modern **OOP** languages do not provide a way to define and control these dependencies. A type system can be used to specify that an observer o observes an instance s of type subject, but the types cannot specify that s will notify o about all changes. To specify this kind of dependencies we need the precise definition of the heap topology between the subjects and the observers. We consider the specification and verification of the observer design pattern in **Section 3.4**.

1.1.4 Verification of security properties

One of the valuable security properties which can be guarantied by heap verification is isolation. The isolation property guarantees that a given piece of code would not affect the rest of the system. For instance, some operating systems enforce isolation between different processes. In such a way a corrupted process with low access rights cannot affect a process with high access rights. Another example is the process isolation in web browsers. Quite often a limited interprocess communication between isolated processes has to be provided. This interprocess communication can create an undesired aliasing between process. This aliasing can be used by one process to affect the state of another process and therefore compromise the isolation. The heap topology specification can be used to prevent such an undesired aliasing between processes.

In **Chapter 8** we consider how our heap verification methodology can be used in each of the above areas.

1.2 Existing approaches for the verification of heap properties

Bellow we propose a classification of existing approaches according to the methods which they use to verify logical properties:

- **Approaches based on decidable procedures.** This group includes all the approaches which for any given verification problem eventually terminate. The group consists of the following sub-groups:

  - *Type systems for alias control.* Approaches from the sub-group use types or types annotations to specify heap properties. Type system rules check the validity of provided annotations.
  - *Approaches based on decidable procedures for a logic theory.* Approaches from this sub-group use the weakest pre-condition calculus or a symbolic computation to reduce a given verification problem to the validation of a formula from a decidable logic.
  - *Approaches based on abstract interpretation.* Approaches from this sub-group use a finite approximation of possibly infinite sets of heap sates. A fix-point computation is used to find an approximation of possible heap states for any point of a given program. Later on these approximations are used to verify validity of heap specifications of the program.

  The main advantage of the decidable procedures is termination. For any given verification problem the procedure will eventually terminate. It is still possible that verification of some programs is not feasible because of a high complexity of the decidable procedure. Nevertheless, it is much better then semidecidable procedures which do not guarantee termination at all.

  The advantage of the decidable procedures which are mentioned above comes at a price. The properties which can be verified by the decidable procedures are limited and do not cover the essential part of the desired heap properties. To be precise, most of the existing decidable procedures are targeted towards the verification of limited properties of various linked lists and trees. Most of the tools are oriented on the verification of shape preservation properties (e.g the reverse of a list is a list) but cannot say much about how exactly the heap changes (e.g how elements of the reversed list are related to the original list).

There is strong evidence that it would be extremely hard or even impossible to overcome these limitations. In **Subsection 7.1.1** we provide references to publications which support this claim.

We provide a detailed overview of approaches based on decidable procedures in **Section 7.1**.

- **Approaches based on interactive theorem prover.** Approaches from this group use various higher order proof assistants. The approaches work in the following way. A user provides a complete proof, including inductive proofs, that a given program satisfies a given specification and the proof assistant checks the correctness of the provided proof.

  These approaches can handle heap properties of arbitrary complexity. On the other hand, a usage of such tool requires a high level of qualification. But even a user of a tool with a proper background and a way of thinking has to invest essential efforts to verify a practical heap problem.

  We provide a detailed overview of approaches based on interactive theorem provers in **Section 7.4**.

- **Approaches based on automatic theorem provers.** Here we include all the approaches which use first order logic (denoted as **FOL**) theorem provers. The main issue for the tools from this group is how to verify properties which use transitive closure and inductive definitions in **FOL**. On one hand these properties are the only way to specify most of the heap structures, but unfortunately verification of such properties requires inductive proofs which cannot be expressed in **FOL**. We classify the approaches from this group according to the way in which they address this issue:

  – *Approaches based on user provided inductive proofs.* Approaches from this sub-group require a user to explicitly provide all necessary inductive proofs and explicitly specify all program points where these proofs are necessary. The validity of the user provided inductive proofs is checked, and the proved **FOL** formulas are added to the corresponding program points.

    The obvious disadvantage of this approach is the extra work which has to be done to perform verification. Another limitation of the approach is that the user has to be qualified enough to be able to produce inductive proofs, and therefore the tool cannot be used by an average software developer. Overall the pros and cons of the approaches from this sub-group are similar to the pros and cons of the approaches based on interactive theorem prover.

– *Approaches based on an approximation of induction in* ***FOL****.* These approaches choose a finite set of **FOL** formulas (whose proofs require the usage of induction) and use them as axioms. The approximation is sound but not complete. Any valid formula can be verified if the chosen set is big enough, otherwise the valid formula would be rejected.

– *Approaches based on user provided updates of a transitive closure.* These sub-group extends the previous one in the following way. In a single program point approaches from this sub-group use an approximation of induction in **FOL** to specify properties of heap structures. On the other hand updates of transitive closures are not inferred by a theorem prover but explicitly provided by a user. In such a way a relation between the heap states before and after a statement execution is always defined precisely, which is crucial for the verification of the effects of the statements.

We provide the detailed overview of the approaches based on automatic theorem provers in **Section 7.2.2**.

We build our methodology on top of approaches based on the user-provided updates of a transitive closure. Let us explain our choice. As it is stated at the beginning of the chapter the goal of this thesis is the automatic formal verification of heap properties of object oriented programs. We cannot use approaches based on decidable procedures because they are not expressive enough. We cannot use approaches based on interactive theorem provers and approaches based on user-provided inductive proofs because they are not automatic enough. Therefore, the only group of approaches which fits our goals are the approaches based on user-provided updates of a transitive closure. Let us consider how they work in more detail.

1.3 Approaches based on user-provided updates of a transitive closure

Let us demonstrate how approaches based on user-provided updates of a transitive closure work in the following example. Instances of class *Node* form a singly linked list. *nextNode* is a field of class *Node* which points to the next node of the list. As an example we consider a specification of the program statement *n.nextNode* = ***null*** where *n* is a variable of type *Node*.

The program statement splits the list (to which n belongs) in two; the first list contains all nodes from the beginning of the list to n, the second list contains all nodes from $n.nextNode$ till the end of the list.

We would like to verify that if the list before the statement execution is acyclic then after the statement execution both output lists are also acyclic. This property can be specified with the help of transitive closure of *nextNode*. Nevertheless, as we mentioned above we cannot express transitive closure in **FOL**. Therefore we approximate transitive closure of *nextNode* in **FOL**.

The approximation of the transitive closure is done in three steps. In the first step we introduce ghost fields `prev` of type reference, `Next`* of type set of references, and `Prev`* of type set of references. Ghost fields are fields used only for specification and can be dropped during compilation. The field `prev` of an object o is supposed to contain the previous element of the list. The field `Next`* of an object o is supposed to contain the set of objects which are reachable from o via *nextNode*. The field `Prev`* of an object o is supposed to contain the set of objects which are reachable from o via `prev`.

In the second step we have to specify the properties of the ghost fields. Some of them can be specified precisely only by means of transitive closure. Since we cannot specify transitive closure precisely in **FOL** we have to provide an approximation of the transitive closure. The provided approximation has to be precise enough to prove the desired properties. To verify the above example we introduce the following properties:

- $\forall o : o.\texttt{prev} \neq \textbf{null} \Rightarrow o.\texttt{prev}.nextNode = o$
- $\forall o : o.nextNode = \textbf{null} \Rightarrow o.\texttt{Next}^* = \varnothing$
- $\forall o : o.nextNode \neq \textbf{null} \Rightarrow o.\texttt{Next}^* =$ $\{o.nextNode\} \cup o.nextNode.\texttt{Next}^*$
- $\forall o : o.\texttt{prev} = \textbf{null} \Rightarrow o.\texttt{Prev}^* = \varnothing$
- $\forall o : o.\texttt{prev} \neq \textbf{null} \Rightarrow o.\texttt{Prev}^* = \{o.\texttt{prev}\} \cup o.\texttt{prev}.\texttt{Prev}^*$
- $\forall o : o \notin o.\texttt{Next}^*$
- $\forall o : o \notin o.\texttt{Prev}^*$

In the above formulas the range of the quantification is non-null references of type *Node*. The first property defines the meaning of `prev`. The next two properties define the meaning of `Next`*. The next two properties define the meaning of `Prev`*. The last two properties guarantee acyclicity of the list.

In the third step we have to define how values of the ghost fields change during the statement execution. In our case the update of ghost fields are specified by the following ghost statements:

- $n.nextNode \neq null \Rightarrow n.nextNode.\texttt{prev} := \textbf{null}$
- $\forall o \in n.\texttt{Prev}^* \cup \{n\} : o.\texttt{Next}^* := o.\texttt{Next}^* \setminus n.\texttt{Next}^*$
- $\forall o \in n.\texttt{Next}^* : o.\texttt{Prev}^* := o.\texttt{Prev}^* \setminus (n.\texttt{Prev}^* \cup \{n\})$

In the above ghost statements we denote field assignment as :=. The ghost statements have to be executed before the program statement *n.nextNode* = ***null***. The first ghost statement specifies that if n is not the last element of the list then `prev` of the next element has to be nullified. The second ghost statement specifies that values $\texttt{Next}^*$ of all nodes which belong to the left tail of the list starting from n have to be decreased by removal of $n.\texttt{Next}^*$. The last ghost statement specifies that values $\texttt{Next}^*$ of all nodes which belong to the right tail of the list starting from the next element of n have to be decreased by removal of $n.\texttt{Prev}^* \cup \{n\}$. We call these updates of ghost field ghost updates.

The preservation of properties of ghost fields by the combination of the assignment to *nextNode* and the ghost updates can be expressed in **FOL** and therefore verified by an automatic **FOL** theorem prover. Since the properties of ghost fields guarantee lists acyclicity we achieve verification of the desired properties in **FOL** without using transitive closure.

The technique sketched above can be used to verify essential heap properties with the help of automatic **FOL** theorem provers. Nevertheless, it has several essential drawbacks.

- **Specification overhead:** By specification overhead we mean the specifications which could be avoided if we used more expressive logics, but have to be provided by a user to enable verification by an automatic **FOL** theorem prover. There are several sources of specification overhead:
  - *transitive closure approximation:* for each usage of transitive closure the user has to provide properties which approximate it. Such an approximation has to be precise enough to verify the desired properties but simple enough to be feasible for automatic theorem provers. The identification of such an approximation can be nontrivial.
  - *ghost updates:* for each update of a field which possibly participates in a transitive closure a ghost update has to be provided. As we have already seen even a single field update can affect the values of the ghost fields of an unbounded set of objects. Quite often ghost fields of all the objects which are reachable from a modified object or which can reach the modified object have to be updated. Therefore even for a simple heap update, the ghost updates can have a significant size.

- **Specification duplication:** By specification duplication we mean the repetition of the same specification in the verified program. There are serval sources of specification duplication:

  - *transitive closure approximation:* the same approximation of the transitive closure can be used to specify different heap structures. The transitive closure approximation duplication consists of the duplication of ghost fields and their properties. For instance, the approximation of transitive closure introduced above could be used to specify a singly or doubly linked lists, cyclic list, a path in a tree, or an acyclic path in a graph. Nevertheless, each of those specifications mentions different fields and therefore has to be repeated multiple times.
  - *ghost updates:* similar updates of fields which participate in an approximated transitive closure require duplication of ghost updates. For instance, every time when we nullify the *nextNode* field we have to repeat the same ghost updates. As we have mentioned in the previous item the same approximation can be duplicated multiple times, which implies that the same ghost updates have to be duplicated by different heap structures but not only by different updates of the same structure. For instance, for each nullification of a field which participates in a list, a tree path, or in an acyclic path in a graph we have to duplicate the same ghost updates.
  - *method and loop invariant specification:* if a ghost update appears in a method body then ghost updates have to be specified in the method post-condition. For instance, if the nullification described above appears in the method ”get tail” then the ghost update has to be duplicated one more time in the method post-condition. If there are several ghost updates in a method body then the method post-condition has to specify their accumulative effect. Such accumulative updates, especially for recursive heap manipulating methods, can be rather complex. Therefore, it is hardly desirable to avoid their duplication. The same applies for the loop invariants of the loops whose bodies modify the heap.

- **Proof duplication:** As it was mentioned above the ghost updates have to preserve the properties of the ghost fields. As it has been mentioned in the previous item there are multiple duplications of the same ghost statement in the source code of the verified program. The same proof of the ghost field properties preservation has to be duplicated for each duplication of the ghost update. Duplication of specifications results in duplication of proofs.

- **Readability of specifications:** We have seen that in order to enable verification the source code has to be annotated with multiple ghost updates; each of them can have significant size. The same applies to method specifications and loop invariants. The specifications of fields are mixed with the specifications of ghost fields. Ghost specification pollutes both source code and specifications. Specification of ghost fields complicates understanding and modification of source code and program specifications.

The heap verification technique which we consider in this thesis is designed with the intention to overcome the mentioned drawbacks.

1.4 Stereotypes

We believe that the previously mentioned problems have the same root. It is the absence of a mechanism which could be used to describe heap properties in an abstract way without mentioning the actual implementation. To fill the gap we introduce the notion of *stereotype*. The aim behind a stereotype is to collect all reusable specifications regarding the heap structure in one entity. We call such entities stereotypes.

Let us consider stereotypes taking the linked list as an example. The specifications which we would like to extract are the description of a sequence of objects. Therefore we call the constructed stereotype `Sequence`. A stereotype consists of *stereotype items* and *stereotype invariants*. Stereotype items describe a heap structure from the perspective of a participating object. Stereotype items are ghost fields which are used for heap specification. The `Sequence` stereotype items are `prev`, `Next`*, `Prev`* which we have already seen in the list specification, plus `next` of type reference. We introduce `next` to avoid the usage of the class field *nextNode*.

If there are several heap structures which can be specified with the same stereotype, for each of them we have to add its own copy of stereotype instances. For instance, if we want to specify with the help of the `Sequence` stereotype a class linked list and a class cyclic list then for both of the classes we have to add ghost fields `prev`, `Next`*, `Prev`*, and `next`. We call this instantiation of the stereotype stereotype slice.

The second part of the stereotype definition is the stereotype invariants. Stereotype invariants specify properties of stereotype items. The `Sequence` stereotype can be received from the list properties which are defined in the previous subsection by substituting *nextNode* by `next`. In this way we receive heap-independent specifications of stereotype items. We call them stereotype invariants because our methodology guarantees their preservation at every program point.

Since a stereotype item is a heap-independent specification there is no direct relation between the heap fields of the specified class and stereotype items. For instance, there is no relation between the class field *nextNode* and the stereotype item `next`. To bind them together we introduce a *glue invariant*. The glue invariant is responsible for establishing a connection between stereotype items and class fields of the specified class. The glue invariant is an extra specification which has to be added to the generic stereotype invariants to adjust them for the verification of a specific class. For instance, to specify the list example above the only glue invariant which has to be added is $\forall o : o.nextNode = o.\texttt{next}$. Since the values of *nextNode* are equal to `next` the set of objects reachable via transitive closure of *nextNode* is equal to the set of objects reachable via transitive closure of `next` and therefore is equal to $\texttt{Next}^*$. Every time when we need the transitive closure of `next` we can use $\texttt{Next}^*$.

We provide definitions of the stereotypes `Sequence` and `Tree` in **Chapter 4**.

Stereotype items and invariants can be used to specify heap structures at a single program point. Nevertheless we need to update values of stereotype items during a program execution. As we have seen these updates can be quite verbose. To avoid duplication of the ghost updates we introduce *stereotype operations*. A stereotype operation aggregates updates of stereotype items. For instance, the ghost updates which we used to specify can be aggregated into a stereotype operation `removeSequenceRelation` which gets one input parameter n of type reference. The input parameter specifies for which object we want to nullify the value of the `next` stereotype item. The operation updates the rest of the stereotype items in such a way that the stereotype invariant is preserved. The preservation of the stereotype invariant is explicitly checked once and for all for each stereotype operation. The specification of the stereotype operation `removeSequenceRelation` can be constructed from the ghost update which we specified in the linked list example by substituting *nextNode* by `next`.

Values of stereotype items can be updated only by a stereotype operation call. For instance, to specify the program statement $n.nextNode = \textbf{\textit{null}}$ we add the stereotype operation call `removeSequenceRelation`(n) to the source code. Since all operations preserve the stereotype invariant we can be sure that the stereotype invariant holds at all program points.

We also use stereotype operation calls to specify class methods and loop invariants. A stereotype operation call can be represented as a pair of pre- and post- conditions which can be added to the method specification. For instance, if the class *Node* of the linked list has a method which consists only of the program statement $n.nextNode = \textbf{\textit{null}}$ then the method can be specified with the help of the same stereotype operation call
`removeSequenceRelation`(n) which we use to specify the ghost updates caused by the statement execution.

Accumulative updates caused by the execution of a heap-manipulating method, especially by a recursive one, can be rather complex. Therefore we introduce the special language for the specification of stereotype operations. The language can be used to describe the most general stereotype operations for a given stereotype. We call such operations universal stereotype transformations. A universal transformation call can be used to specify precisely an arbitrary heap manipulating method. We provide definitions of universal transformation for `Sequence` and `Tree` stereotypes in **Appendix A** and **Appendix B**, respectively.

The stereotype-based approach addresses the drawbacks of the approaches based on user-provided updates of a transitive closure in the following way:

- **Specification overhead:**

  To enable automatic verification we still need redundant descriptions of ghost fields, their properties, and ghost updates. Nevertheless, we believe that most of such specifications can be provided in a predefined standard stereotype library. For instance, in this thesis we provide descriptions of `Sequence` and `Tree` stereotypes. These stereotypes from the standard library can be used to specify a program with minimal overhead. To specify a given program with the help of stereotypes from the standard stereotypes library the user has to provide the glue invariants for each class and the stereotypes operations calls for each ghost update. In **Chapter 5** we demonstrate how `Sequence` and `Tree` stereotypes can be used to verify various design patterns and heap structures.

- **Specification duplication:**

  The stereotype usage completely prevents duplication of ghost fields, properties of ghost fields, and ghost updates. Duplication of ghost fields and their properties is prevented by usage of stereotype items and stereotype invariants respectively. Duplication of ghost updates, method specifications, and loop invariants is prevented by stereotype operations.

- **Proof duplication:**

  As it has been mentioned above the reason for proof duplication is specification duplication. Every ghost update duplication results in the duplication of the proof of the preservation of the ghost field properties. In the presence of stereotypes all ghost updates are packed into stereotype operations. For each of the operations preservation of the stereotype invariants (which aggregate properties of ghost fields) the proof is done only once when the operation is defined. In the operation call the validity of the stereotype invariant is not proved but assumed.

- **Readability of specifications:**

In the absence of stereotypes it is typical that class fields are mixed with ghost fields, and source code is mixed with ghost updates. The situation is even more problematic if a class participates in several heap structures and we use different ghost fields to specify properties of different heap structures. In such case it can be problematic to identify which ghost fields or ghost updates correspond to which heap structure. On the other hand, in the presence of stereotypes we can:

– distinguish between source code and ghost updates; ghost updates are represented by stereotype operation calls while source code is represented by direct field updates and method calls.

– distinguish between ghost updates which correspond to different heap structures.

– use the same stereotype operations for both ghost updates and the specification of methods and loop invariants.

Among other factors specification duplication can have a negative influence on readability of specifications. For example, assume that several classes use ghost fields to specify the same behavior of sequence, then a reader of the classes has to put significant efforts to understand that all these specifications have the same meaning. On the other hand, with the stereotype approach we can immediately see when several classes participate in the same stereotype. There is a similar situation with ghost updates. It can be a nontrivial task to recognize that several independent groups of ghost updates have the same meaning. On the other hand it becomes obvious when we use the same stereotype operation for these updates.

Additionally, dealing with the above problems, stereotypes have the following attractive properties:

- The stereotypes based approach splits the specification process into two independent stages. The first one is the development of stereotypes. The second one is usage of stereotypes for the specification of heap structures. We assume that most of the time developers work on the second stage. We also assume that most of the stereotypes can be provided as a standard library. Since the usage of the stereotypes, in contrast to their development, does not require a deep logical background we believe that in most cases stereotypes can be used by developers who do not have a deep background in logic.

- Another advantage of the stereotype based approach is the possibility to specify the switch of an object's field from one heap structure to another during a program execution. This mechanism significantly improves reusability of stereotypes. In case we need several stereotypes to describe different aspects of a data structure, which is quite common, we can reuse stereotypes with minimal additional efforts. Let us consider the following situation: there is a class C with a reference field f. Let us also assume that during the program execution an instance o of class C participates either in a tree or in a cyclic list. Here we have a situation when the same location $o.f$ changes its role during the program execution. Significant efforts have to be made to specify this role change with just ghost fields and ghost updates. On the other hand the description of the class field role change as a combination of stereotypes is relatively straightforward.

The role change described above occurs in the implementation of the priority inheritance protocol [132] (denoted as **PIP**). In **Section 5.4** we consider how **PIP** can be specified with the help of `Sequence` and `Tree` stereotypes. We use this specification to verify **PIP** in the verification language **Boogie** [75]. The verification is described in **Subsection 5.5.2**. According to our knowledge it is the first automatic verification of **PIP**.

1.5 The project context

We developed the stereotype methodology during the MOBIUS EU project. The project was dedicated to the Proof-Carrying Code (abbreviated as **PCC**) [110]. **PCC** concepts can be applied only for a verified source or byte code. Therefore a part of the project was dedicated to overcome some of the limits of automatic software verification. In the context of this part of the project we addressed two issues of automatic software verification:

1. verification of heap structures.

2. checking of the feasibility of pure-method specifications.

To address the first issue we proposed **UTT**; a combination of an ownership-based type system and a data flow analysis. The result of our research was presented at OOPSLA 2007 [104]. **UTT** and several of its extensions dedicated to specification inference, error explanations, and runtime checking were implemented during multiple student projects at ETH Zurich [139, 128, 96, 115]. The soundness proof of **UTT** can be found in a technical report [103].

Pure-methods provide a natural way to introduce a semantics of domain specific functional symbols. Instead of introducing axioms, a specification developer can provide a pre- and post- condition for a pure method and then use it in other method specifications. Nevertheless, if the method specifications are infeasible then the method usage can compromise soundness of the verification process. To prevent this we develop a methodology for checking the feasibility of pure methods. The results were presented at FM 2008 [124] and IJCAR 2008 [38]. The soundness proof of the methodology can be found in a technical report [125].

It needs to be mentioned that both proposed techniques have some limitations which prevent their usage in real life projects. **UTT** can be used only to deal with heap structures which are tree shaped. Another limitation of **UTT** worth mentioning is that it cannot be used for verification of heap structures separation. The root of these limitations is the absence of precise heap topology information. The feasibility validation methodology which we proposed relies on the substantial usage of quantifier alternation and inductive proofs to check recursive functions. On the other hand, as we have already mentioned heap specifications widely use transitive closure or recursive functions. Therefore the methodology cannot be directly applied to check the feasibility of heap specifications.

The desire to overcome these limitations resulted in the development of the stereotype methodology. In contrast to **UTT**, the stereotype methodology can deal with arbitrary heap structures, not only trees, and can be used to verify any kind of heap properties. As far as feasibility checking is concerned it is directly built into the stereotype methodology and guaranteed by construction.

1.6 Outline

The thesis is organized in the following way. In **Chapter 2** we introduce a motivating example, **PIP**. Then in **Chapter 3** we provide the definition of stereotype and basic stereotype operations. The provided definitions are demonstrated on examples of `Sequence` and `Tree` stereotypes in **Chapter 4**. In **Chapter 5** we provide several examples of the stereotype based verification including **PIP**. A methodology for constructing stereotype operations is provided in **Chapter 6**. In **Chapter 7** we discuss related work. **Chapter 8** concludes the thesis by summarizing the approach, sketching avenues for future development, and describing how various areas of software verification, software engineering, and theorem proving could benefit from the stereotype methodology. **Appendix A** and **Appendix B** contain the complete specification of `Sequence` and `Tree` universal transformations.

Chapter 2

Motivating example: priority inheritance protocol

In this chapter we consider the algorithm for a *priority inheritance protocol* [132] (we refer to it as **PIP**). **PIP** has a non trivial heap topology and therefore we use it to evaluate the stereotype methodology. The complete specification of **PIP** is provided in **Section 5.4**. Here we consider only how an implementation of **PIP** works.

We choose **PIP** as a motivating example for the following reasons:

- **PIP** is widely applicable in the area of real-time operating systems. In 1997, the Mars Pathfinder mission nearly failed because of an undetected priority inversion. Such a situation might have been avoided had the designers of the rover used **PIP** in a proper way [73].

- During an execution of a **PIP** implementation a nonstandard heap structure is created. This heap structure combines the behavior of both cyclic lists and trees. It would be hard to describe the desired structure as a combination of tree and cyclic list **ADT**s. On the other hand it is relatively straightforward to describe it as a combination of the corresponding stereotypes (see **Chapter 5**).

- An important part of the **PIP** behavior is implicit updates. Namely, it is possible that an invariant of an object o can be broken by another object o' and there is no reference from the object o to the object o'. In general there could be an unbounded number of objects like o'. This pattern is a hard problem for software verification. In **Chapter 5** we demonstrate how we can use the stereotype approach to specify and verify functional properties of **PIP**.

PIP works with two kind of entities: *tasks* and *resources*. A task corresponds to a sequence of operations that are performed concurrently. Tasks use resources for their work. Before using a resource a task has to *acquire* it. After using the resource the task has to *release* it. Between acquiring and releasing the resource the task *owns* the resource, which means that the task has an exclusive read/write control over the resource and can arbitrarily change it. A task can own an unbounded number of resources, but each resource is owned by at most one task (some resources are unowned). If a task t tries to acquire a resource r owned by another task then task t waits until the owner of r releases it. We say that the task t is *blocked by* the resource r. A resource can block an unbounded number of task, but each task is blocked by at most one resource (some tasks are unblocked). It is possible that the blocked-by relation can form a cyclic dependence. Such a situation is denoted as a deadlock. For example, let us consider the following situation. A task t_1 owns a resource r_1 and a task t_2 owns a resource r_2. If t_2 requests r_1 and t_1 requests r_2 then both t_1 and t_2 are deadlocked (see Figure 2.1).

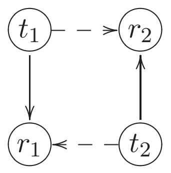

$$\begin{tikzcd} t_1 \arrow[r, dashed] \arrow[d] & r_2 \\ r_1 & t_2 \arrow[l, dashed] \arrow[u] \end{tikzcd}$$

Figure 2.1: An example of a deadlock which can arise in a concurrent environment. Here we denote tasks and resources as circles, owned-by relation as solid arrows, and attempts to acquire resource as dashed arrows.

In real-time applications different tasks have different importance. Some of them can be delayed for some time. On the other hand there are mission critical tasks which have to be executed under strict time constraints. To reflect this fact we associate each task with a priority. An expected behavior is that a task with a higher priority is given execution time preferentially, compared to a task with a low priority. On the other hand a high priority task *preempts* a low priority task. Unfortunately, the need to share resources between tasks can result in an unexpected behavior. For instance, a high priority task $t_{\mathbf{H}}$ is blocked by a resource r, and resource r is owned by a low priority task $t_{\mathbf{L}}$. If a middle priority task $t_{\mathbf{M}}$ is executed concurrently with $t_{\mathbf{H}}$ and $t_{\mathbf{L}}$ (see Figure 2.2) then, since $t_{\mathbf{M}}$ has a higher priority than $t_{\mathbf{L}}$, $t_{\mathbf{M}}$ preempts $t_{\mathbf{L}}$. If there was no $t_{\mathbf{M}}$, execution time preference would be given to $t_{\mathbf{L}}$. In such a case $t_{\mathbf{L}}$ would be completed as soon as possible, and execution time preference would be given to $t_{\mathbf{H}}$. But since there is $t_{\mathbf{M}}$ and $t_{\mathbf{H}}$ is waiting for $t_{\mathbf{L}}$, $t_{\mathbf{M}}$ is implicitly preempts $t_{\mathbf{H}}$. At the end we have a situation in which a task with a high priority is delayed for an unbounded time by a task with a lower priority. Such a situation is called an *unbounded priority inversion.* It can result in a time constraint violation for a mission critical task, like it happened with the Mars Pathfinder during collecting of a meteorological data on Mars [73].

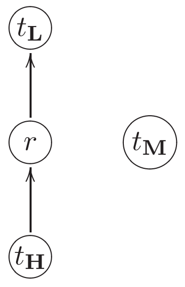


Figure 2.2: An example of an *unbounded priority inversion* which can arise in a concurrent environment. Here we denote tasks and resources as circles, and owned by and blocked-by relation as solid arrows.

One way to avoid the priority inversion is to use **PIP**. The main idea behind **PIP** is to use dynamic priority adjustments. Now each task has two priorities: a static and a dynamic one. Initially the static priority is equal to the dynamic priority. Let us consider the situation when a task t acquires a resource r owned by an unblocked task t'. To avoid a situation like we have seen above we temporally increase the dynamic priority of the t' to the maximum of dynamic priorities of t and t'. Once t' releases r, we recalculate its dynamic priority as maximum of the static priority of t' and dynamic priorities of tasks blocked by t'. We can't just drop back the priority of t' to its original priority, since there could be another task t'' which is also blocked by t' and has the same dynamic priority as t

Let us now consider the case when t' is blocked. In such a case we have to take care of tasks which are reachable by inversion of the blocked-by relation. For instance let us consider the following situation, t' is blocked by t_1, t_1 is blocked by t_2, …, t_{n-1} is blocked by t_n. If t' acquires or releases a resource it potentially can change its dynamic priority. Since t' is blocked by t_1 change of the dynamic priority of t' can affect the priority of t_1. Change of the priory t_1 can affect the priority of t_2, and so on until priority of the t_n is affected.

The main property which we expect to hold for the dynamic priority of a task t is that the dynamic priority of the task t is equal to the maximum of the static priority of the task t and dynamic priorities of task blocked by the task t.

Let us now look in more detail into a possible **Java** implementation of **PIP** (see Figure 2.3). To simplify the implementation we use the same class $PIPNode$ to represent both tasks and resources. The field *blockedBy* is used to represent the blocked-by relation for tasks and the owned-by relation for resources. We say that a node n is in the blocked-by relation with a node n' if and only if $n.blockedBy = n'$. The $defaultPriority$ and the $currentPriority$ fields represent the static and the dynamic priorities of a node. The multiset *priorities* is an auxiliary field which we use to facilitate priority updates. We say that the *priorities* field of a node n is in a valid state if and only if *priorities* contains exactly the dynamic priorities of all nodes blocked by n excluding zeros. It is easy to see that if the *priorities* multiset is in a valid state then it can be used to compute the dynamic priority of the node in the following way: $currentPriority = max(defaultPriority, max(priorities))$.

The **PIP**'s functionality consists of one constructor and three methods, one is an auxiliary and two are provided to clients.

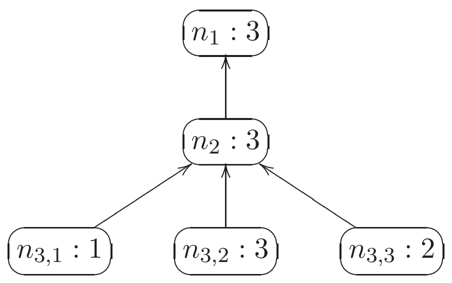
The constructor creates a singleton **PIP** node. The only input parameter of the constructor is `priority`. It is used to set both the current and the default priority. Also the constructor sets *blockedBy* to **null** and *priorities* to an empty multiset.

The auxiliary method is $updatePriorities$. It is used to restore *priorities* and *currentPriority* to valid states. There are three possible ways in which the *priorities* multiset of a node n can become invalid: resource blocking, resource releasing, and change of the dynamic priority of a node blocked by the node n. We can see that in each of these cases it is enough to add not more than one number to and remove not more than one number from *priorities* to restore validity of *priorities*. That is why $updatePriorities$ gets exactly two input parameters. The first one, $from$, represents the value that has be removed from and the second one, *to*, represents the value that has to be added to the *priorities* multiset to restore its validity. It is possible that $from$ or *to* is equal to 0; in this case the corresponding parameter doesn't affect *priorities*. As soon as validity of *priorities* is restored we use it to compute a new value of the dynamic priority. It is possible that a receiver of the *updatePriorities* (let's denote it as n) is blocked by another node (let's denote it as n'). If it is so then a change of the dynamic priority of the n invalidates the *priorities* of the n'. That is why we conclude the implementation of the *updatePriorities* with a recursive call with the receiver n'. The only change which happens with the set of nodes blocked by n' is a change of the dynamic priority of n. The effect of this change on the *priorities* multiset of n' is equivalent to the addition of a node with the dynamic priority equal to the new dynamic priority of n and removal a node with the dynamic priority equal to the old dynamic priority of n. That is why the first parameter of the *updatePriorities* call is the old dynamic priority of n and the second one is the new dynamic priority of n.

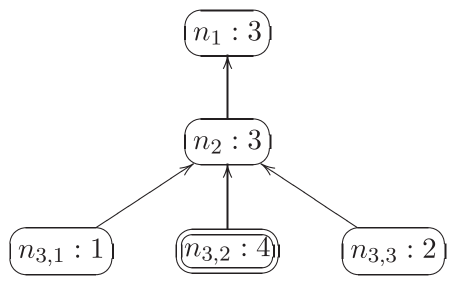
Let us now consider how updatePriorities works on the example which is presented on Figure 2.4. Here we depict nodes as rounded rectangles and blocked-by relation as arrows. Inside a rectangle we can see the name of the node to the left of the colon, and the dynamic priority to the right of it. We mark with the double box the receiver of a currently executed method. Let us consider what happens if we make the following method call $n_{3,2}.updatePriorities(0, 4)$. On Figure 2.4(a) you can see the state before

```
class PIPNode{
  PIPNode blockedBy;
  int defaultPriority;
  int currentPriority;
  MultiSet⟨int⟩ priorities;

  PIPNode(int priority){
    blockedBy = null;
    defaultPriority = priority;
    currentPriority = priority;
    priorities = ∅;
  }

  void updatePriorities(from: int, to: int){
    oldCurrentPriority int;
    oldCurrentPriority = currentPriority;

    if (from > 0)
      priorities = priorities\{from};
    if (to > 0)
      priorities = priorities∪{to};

    currentPriority = max(defaultPriority, max(priorities));
    if(blockedBy ≠ null && oldCurrentPriority ≠ currentPriority)
      blockedBy.updatePriorities
        (oldCurrentPriority, currentPriority);
  }

  void release(n: PIPNode){
    n.blockedBy = null;

    if (n.currentPriority ≠ 0)
      updatePriorities(n.currentPriority, 0);
  }

  void acquire(n: PIPNode){
    if(n.blockedBy = null){
      n.blockedBy = this;
      if (n.currentPriority ≠ 0)
        updatePriorities(0, n.currentPriority);
    } else {
      this.blockedBy = n;
      if (currentPriority ≠ 0)
        n.updatePriorities(0, this.currentPriority);
    }
  }

}
```

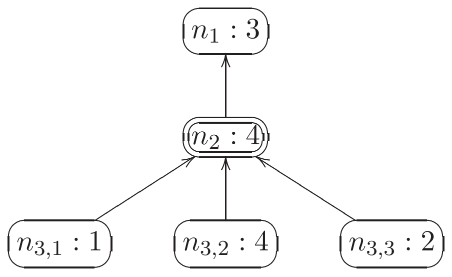
Figure 2.3: Implementation of the priority inheritance protocol.



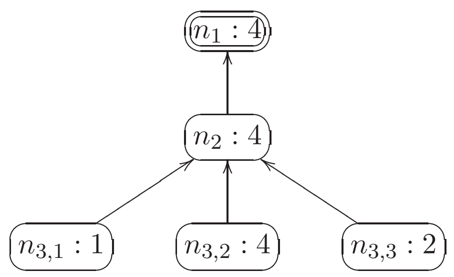
(a) Before execution of the updatePriorities method on $n_{3,2}$



(b) During execution of the updatePriorities method on $n_{3,2}$



(c) During execution of the updatePriorities method on n_2



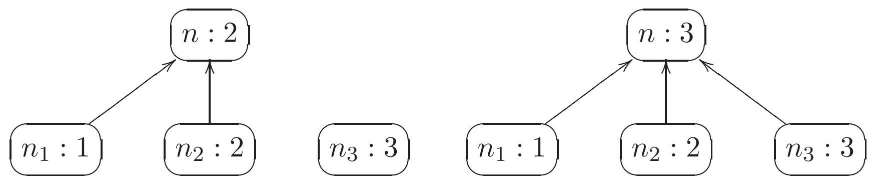
(d) During execution of the updatePriorities method on n_1

Figure 2.4: An example of updatePriorities method call. Here we depict nodes as rounded rectangles and blocked-by relation as arrows. Inside of a rectangle we can see the name of the node to the left of the colon, and the dynamic priority to the right of the colon. We mark with the double box the receiver of a currently executed method.

the method call. The first part of the method recalculates and sets the new value of the dynamic priority of $n_{3,2}$ (see Figure 2.4(b)). After this it makes the recursive call to update the priority of n_2 (see Figure 2.4(c)). After updating its dynamic priority n_2 propagates the call to n_1 (see Figure 2.4(d)). Since n_1 is unblocked, the method's execution terminates after updating the value of the dynamic priority of n_1.

The **PIP** implementation provides two methods to a client: *release* and *acquire*. The method *release* is used when a node n is blocked by a node n' and we want to release the node n. To do it we set *blockedBy* field of the n to *null*. As the result, if the dynamic priority of the n is not equal to 0 we invalidate the state of the *priorities* of n'. To restore validity of the *priorities* of n' we use *updatePriorities*. The only change which happens with the set of nodes blocked by the node n' is the removal of node n. That is why the first parameter of the *updatePriorities* call is the dynamic priority of the removed node n and the second one is 0 (we do not add any node).

The method *acquire* is used when a node n' tries to block a node n. Depending on whether a node n is blocked or not there are two possible outcomes of the method. If the node n is not blocked then node n' will block it. Since the set of nodes blocked by the node n' is changed we call *updatePriorities* to restore validity of the *priorities* of n'. The only change which happens with the set of nodes blocked by the node n' is the addition of the node n. That is why the first parameter of the *updatePriorities* call is 0 (we do not remove any node) and the second one is the dynamic priority of the added node n. Otherwise, if the node n is blocked then we perform a dual action, namely n blocks n'. Since the set of nodes blocked by the node n is changed we call *updatePriorities* to restore validity of the *priorities* of the n. The only change which happens with the set of nodes blocked by the node n is addition of the node n'. That is why the first parameter of the *updatePriorities* call is 0 (we do not remove any node) and the second one is the dynamic priority of the added node n'. In the last case it is possible that n is transitively blocked by n'. If it is so then the *acquire* method's execution creates a deadlock.



(a) Before execution of the *acquire* method, or after execution of the *release* method

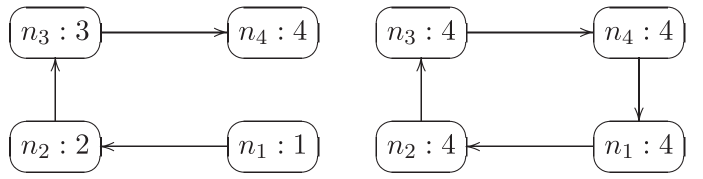
(b) After execution of the *acquire* method, or before execution of the *release* method

Figure 2.5: An example of *release* and *acquire* method calls on an acyclic graph. Here we depict nodes as rounded rectangles and blocked-by relation as arrows. Inside of a rectangle we can see the name of the node to the left of the colon, and the dynamic priority to the right of it.

Let us now consider examples that illustrate how *acquire* and *release* works. There are mainly two distinct cases which we should consider: a call on an acyclic and on a cyclic graph.

On Figure 2.5 you can see an example of a call on an acyclic graph. We call it acyclic because we do not create a loop by calling *acquire*. In this example n acquires n_3 by executing $n.acquire(n_3)$ (see Figure 2.5(a)). There is the result of the execution of *acquire* on Figure 2.5(b). Since n now owns n_3 with the dynamic priority 3, it has to increase its own priority to 3.

The same example can be used to demonstrate how *release* works. If we execute $n.release(n_3)$ on the state pictured on Figure 2.5(b) we will get the state pictured on Figure 2.5(a). Here we can see that after releasing of n_2 node n drops back it priority to 2. We should notice that such dropping back of the priority is not always the case. If priority of n_2 or n_1 also was 3, n would keep it priority even after releasing n_3.



(a) Before execution of the *acquire* method, or after execution of the *release* method

(b) After execution of the acquire method, or before execution of the release method

Figure 2.6: An example of *release* and *acquire* method calls on a cyclic graph. Here we depict nodes as rounded rectangles and blocked-by relation as arrows. Inside of a rectangle we can see the name of the node to the left of the colon, and the dynamic priority to the right of it.

On Figure 2.6 you can see an example of a call on a cyclic graph. We call it cyclic because we do create a loop by calling *acquire*. In this example n_4 acquires n_1 by executing $n_4.acquire(n_1)$ (see Figure 2.6(a)). There is the result of the execution of *acquire* on Figure 2.6(b). Since n_1 is already blocked-by another node, the execution of *acquire* results in blocking of n_4 by n_1. This blocking creates a cyclic dependence between nodes $n_1, \ldots, n_4$. After creating this loop, *acquire* calls updatePriorities which increases priorities of all nodes participating in the list up to 4.

The same example can be used to demonstrate how *release* works. If we execute $n_1.release(n_4)$ on the state pictured on Figure 2.6(b) we will get the state pictured on Figure 2.6(a). Here we can see that releasing of n_4 brakes the loop all nodes which participate in the loop drop back their priorities.

Up to now we ignored issues related with possible interference between **PIP** methods during their parallel execution. Nevertheless, it is possible that such an interference can result in a race condition. One possible way to avoid such race conditions is to use a global lock. Any **PIP** method acquires this lock before and releases it after the execution. We can easily see that this modification of the **PIP** implementation behaves as if it was executed in a sequential but not in a concurrent way. This observation justifies that from here on we treat the **PIP** implementation as a sequential program.

Let us consider the properties of **PIP** which we want to verify. The main property which we want to verify is that value of the field $currentPriority$ of all instances of class $PIPNode$ are correct. The value of the field $currentPriority$ of a node n is correct if and only if $currentPriority$ of n is equal to the maximum of $defaultPriority$ of n and of $currentPriority$ of nodes blocked by n.

To verify the above property we have to specify and verify **PIP**'s heap topology. The heap topology of **PIP** is formed by the *blockedBy* field. An example of a heap layout of **PIP** is provided in Figure 2.7. If there are no deadlocks then the *blockedBy* forms a tree relation. By acquiring

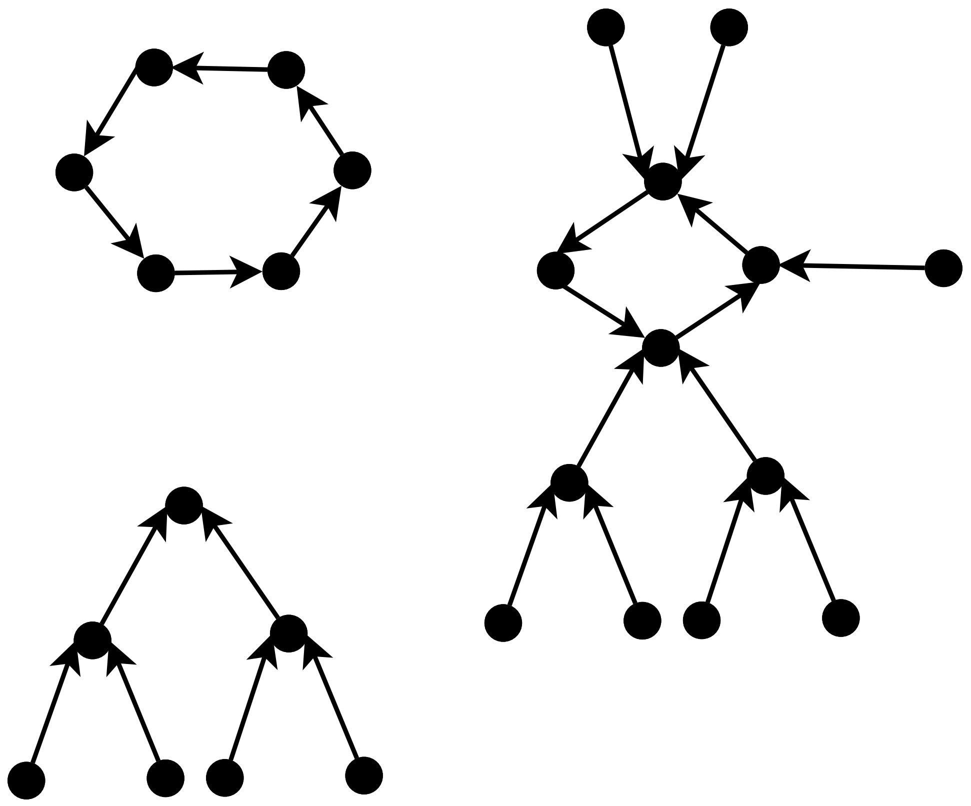


Figure 2.7: The figure depicts a heap layout for the PIP example. PIP nodes are denoted as black circles. Black arrows denote the blockedBy field.

and releasing nodes we add and remove sub-trees. On the other hand if a node n attempts to acquire a node n' which is already locked by one of n's descendants then the deadlock is created.

We provide the complete specification of **PIP** in **Section 5.4**. We describe verification of **PIP** in the verification language **Boogie** [75] in **Subsection 5.5.2**.

Chapter 3

Stereotype-based verification methodology

In this chapter we are going to describe our stereotype-based verification methodology. The methodology introduces a significant number of notions. Therefore on Figure 3.1 we show the **UML** diagram for stereotype-based verification and object oriented programming related entities. This diagram can be used as a road map for this section. In the diagram we use the following notations: boxes depict entities; arrows depict relations between entities; arrows with the black diamond arrowhead depict composition relations; arrows with the white triangle arrowhead depict specialization relations.

The figure is split in two parts which are bordered by dashed rounded rectangles. The upper rectangle borders entities related to object oriented programming. The bottom rectangle borders entities related to stereotype-based verification. The chapter is dedicated to the introduction and illustration of these notions.

The chapter is organized in the following way. We begin with a motivating example in **Section 3.1**. **Section 3.2** is dedicated to the description of stereotypes. A stereotype consists of stereotype items, which are introduced in **Subsection 3.2.1**, and stereotype invariants which are introduced in **Subsection 3.2.2**. These notions are combined into stereotypes in **Subsection 3.2.3**. **Subsection 3.2.1** also describes how an instantiation of a stereotype can be constructed. We call this instantiation a stereotype slice.

Section 3.3 introduces stereotype operations which are used to update values of stereotype slices. In **Subsection 3.3.1** we introduce basic stereotypes operations. Basic stereotype operations are the simplest stereotype slice transformations which preserve the stereotype invariants. In **Subsection 3.3.2** it is described how basic stereotype operations can be used to construct more advanced operations.

Section 3.4 is dedicated to the description of how stereotype slices and operations can be used to specify source code. In **Subsection 3.4.1**

we describe how stereotype slices can be declared in a program. **Subsection 3.4.2** describes how stereotype operations can be used to specify updates of stereotype slices. **Subsection 3.4.3** introduces glue invariants. Glue invariants bind together stereotype items and fields of the program heap. In **Subsection 3.4.4** we describe how stereotype operations can be used to specify properties of class methods. We conclude by describing in **Subsection 3.4.5** how behavioral class invariants can be used on top of heap topology properties.

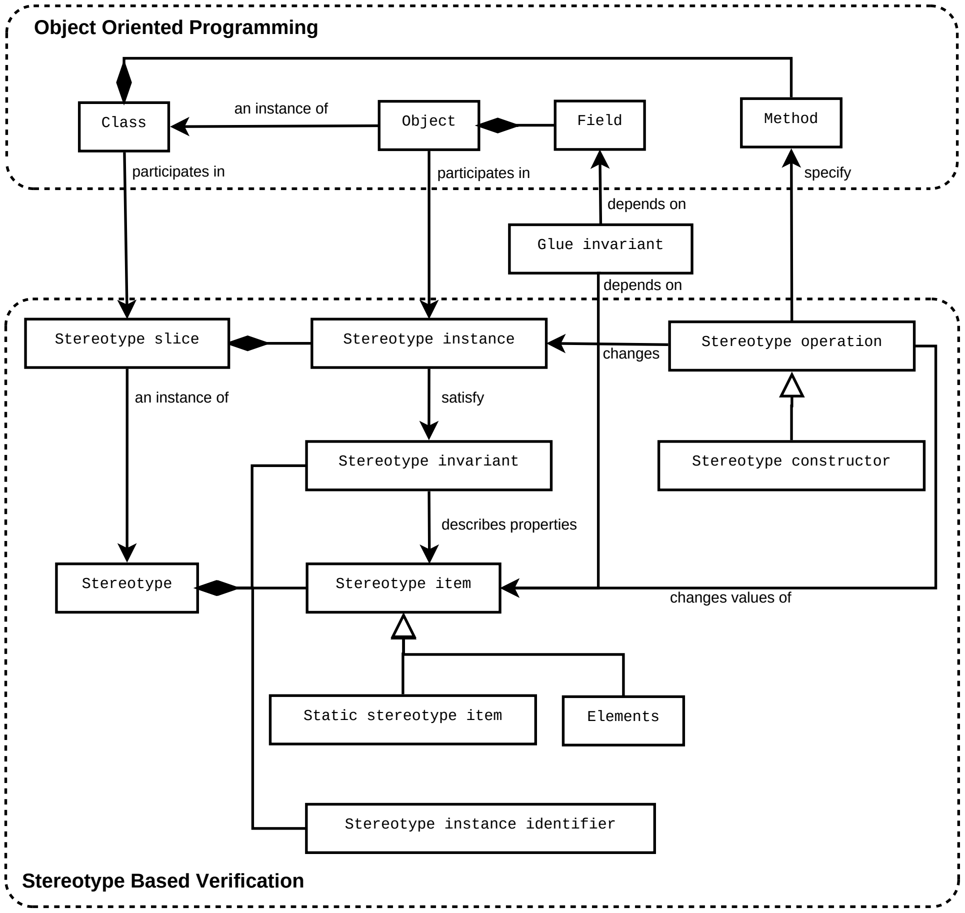


Figure 3.1: **UML** diagram for stereotype related entities.

3.1 Motivating example

As we mentioned above one of the main motivations for using stereotypes is the reusability of specifications and proofs. The intention of this section is to demonstrate this aspect of stereotype usage. To achieve this we consider three design patterns and outline a behavioral aspect which they share.

In the next sections this common behavioral aspect is formalized as the `RelationInversion` stereotype. Descriptions of these design patterns are taken from [50]

The first design pattern which we consider is the observer pattern. The observer pattern is used to define a one-to-many dependency between objects so that when one object changes its state, all its dependents are notified and updated automatically. The key classes in this pattern are **Subject** and **Observer** (see Figure 3.2). A subject may have any number of dependent observers. An observer can attach to and detach from a subject using methods $Attach(Observer)$ and $Detach(Observer)$. All observers are notified via method $Update()$ whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subject's state.

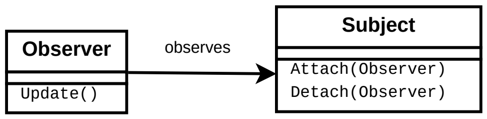


Figure 3.2: *Observer pattern.* Here we use the standard **UML** notation. We represent classes as rectangles. There is a class name at the top of the box. Methods of the class are listed in the lower part of the box. We depict semantic relations between classes as arrows. Each semantic relation is labeled with a short description.

The second design pattern is the iterator pattern. The iterator provides a way to access the elements of an aggregate object sequentially without exposing its underling representation. An iterator object is responsible for keeping track of the current element. It knows which elements have been traversed already. A typical instantiation of the iterator pattern is the list iterator (see Figure 3.3). Here **ListIterator** plays the role of an iterator and **List** plays the role of an aggregate object.

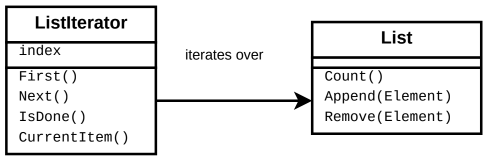


Figure 3.3: *Iterator pattern.*

The third design pattern is the mediator pattern. The motivation behind this pattern is the following. Object-oriented design encourages the distribution of behavior among objects. Such a distribution can result in an object structure with many connections between objects; in the worst case, every object ends up knowing about all other objects. It is hard to understand and maintain such a program. Modification of a single class can require to modify other related classes. We can avoid these problems by encapsulating the communication between objects in a separate **Mediator** object. The mediator serves as intermediary that keeps objects in the group from referring to each other explicitly. We denote an object from the group as **Colleague**. The colleagues only know the mediator, thereby reducing the number of interconnections (see Figure 3.4).

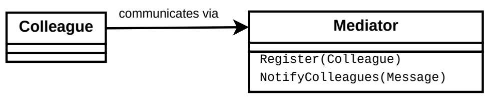


Figure 3.4: *Mediator pattern.*

Let us consider typical usages of these design patterns:

- *Observer*: One of the observers changes the state of the subject. In response to this the subject notifies all other observers about the update.
- *Iterator*: A list iterator modifies the list. Depending on the concrete implementation some of iterators can be invalidated and cannot be used anymore. For instance in the **Java** standard library a removal of a list element via an iterator invalidates all other integrators.
- *Mediator*. A colleague can send a message via the mediator. The mediator forwards the message to all subscribed colleagues.

The above examples can be described in an abstract uniform way. The description is based on a unary function f which maps an object reference to another object reference. f has the following meaning for the described above design patterns:

- *Observer*: f maps an observer o to the subject $f(o)$ which is observed by o.
- *Iterator*: f maps an iterator o to the list $f(o)$ which is iterated over o.
- *Mediator*: f maps a colleague o to the mediator $f(o)$ via which o communicates with other colleagues.

With the help of f the above examples can be described as the following sequence of steps:

- an object o updates object $f(o)$
- $f(o)$ updates all objects which belong to the set $f^{-1}(f(o))$

We can see that to specify the above examples f and f^{-1} have to be specified. The specification of f and f^{-1} may be challenging for the following reasons.

First of all the value of f and therefore f^{-1} are dynamic; they can change during the program execution. Let us consider how it happens in each of the design patterns

- *Observer*: Attachment of an observer o to a subject o' leads to the addition of $\langle o, o' \rangle$ to f. Detachment of an observer o from a subject o' leads to the removal of $\langle o, o' \rangle$ from f.

- *Iterator*: Creation of an iterator o by a list o' leads to the addition of $\langle o, o' \rangle$ to f.

- *Mediator*: Registration of a colleague o for communication via a mediator o' leads to the addition of $\langle o, o' \rangle$ to f.

Therefore a proper specification of f has to include a specification of possible updates of f.

Secondly, another circumstance which complicates the specification of f and f^{-1} is a possible interaction between design patterns instances. The following example demonstrates how this can happen. Let us assume that there are two lists o'_1 and o'_2, and o'_1 creates a new iterator o. The creation of the iterator by the list o'_1 should not affect another list o'_2. Nevertheless, since f is defined globally on all references in the heap we cannot take this property for granted. Therefore we include disjointness between parts of f which corresponds to different design pattern instances into the specification of f.

We can see that the specification of f is reused by all of the above design patterns. On the other hand, the specification of f requires significant efforts. Therefore, instead of specifying f for each of the above examples it can be specified once and used when needed. In the next section we show how f can be specified as the `RelationInversion` stereotype. Then, in **Section 3.3** we specify updates of f as `RelationInversion` stereotype operations. To sum up, in **Section 3.4** we demonstrate how the `RelationInversion` stereotype can be used to specify the three design patterns which we considered in this section.

3.2 Stereotypes

In this section we introduce stereotypes. A stereotype is an abstract description of a specification aspect. Function f from the previous section is an example of such a description. We formalize f as `RelationInversion` stereotype.

A stereotype definition consists of stereotype items and stereotype invariants. Stereotype items represent the state of the specification aspect. For instance, the stereotype items of the `RelationInversion` stereotype are a representation of the function f. Stereotype invariants describe properties of the captured specification aspect in terms of specification items. For instance, the stereotype invariants of `RelationInversion` describe properties of the function f.

In order to use a stereotype in source code specifications it has to be instantiated. We call an instantiation of a stereotype stereotype instance. For instance, in order to specify the three design patterns above we have to introduce three stereotype instances of the stereotype `RelationInversion`.

A stereotype slice of a stereotype St consists of an instantiation of the stereotype items and an instantiation of the stereotype invariants. An instantiation of the stereotype items are ghost fields which are added to all classes which are specified by the stereotype slice. A ghost field is a field which is used for specifications only and can be removed during the program compilation. An instantiation of stereotype invariants consists of two parts. The first part is the same for all stereotypes. It describes a partition [20] of the set of heap references. This partition is used to guarantee disjointness between various instances of a design pattern or a data structure which is specified by the stereotype slice. We call each part of the partition a stereotype instance. Later on we will provide an example which explains the difference between a stereotype instance and a stereotype slice. The second part of the instantiation of stereotype invariants is constructed from the stereotype invariants by replacing stereotype items with the corresponding ghost fields and by restriction range of quantification on values from the same stereotype instance.

3.2.1 Stereotype items

Definition 1 (Stereotype item)**.** *A stereotype item is a pair of a stereotype item name and a type.*

In this thesis we use the following types for stereotype items:

- ref: reference type.
- Reg: set of references or region type.
- $ref \to ref$: a function from a reference to a reference.
- $ref \to Reg$: a function from a reference to a region.
- $ref^2 \to ref$: a function from a pair of references to a reference.
- $ref^2 \to Reg$: a function from a pair of references to a region.

A value of a region type is a set. We use standard set operations and predicates: $\cup$ union, $\cap$ intersection, $\overline{S}$ complement of a set S, $=$ set equality, $\#$ set disjointness. For technical reasons we define a singleton set creation in a slightly unusual way. If o is a reference and o is not equal to null then $\{o\}$ is a singleton set which contains o, otherwise $\{o\} = \varnothing$. This non standard definition helps keeping specifications shorter. We denote the result of the application of the function f to a value v as $f[v]$. We denote the image of a set S under function f as $f(S)$.

For example, the `RelationInversion` stereotype has the following two items: *sink* of type *ref* and *Source* of type *Reg*. The names of stereotype items of reference type begin with a small letter (e.g., *sink*) and names of stereotype items of region type begin with a capital letter (e.g. *Source*). Let us explain the meaning of these stereotype items for the design patterns from the previous section.

- For an instance of the observer design pattern *sink* refers to the subject and *Source* contains the set of observers.
- For an instance of the iterator design pattern *sink* refers to the list and *Source* contains the set of iterators.
- For an instance of the mediator design pattern *sink* refers to the mediator and *Source* contains the set of colleagues.

As you can see we explained the meaning of stereotype items *sink* and *Source* in terms of design pattern instances. The notion of a design pattern or data structure instance is hardcoded into stereotypes. To formalize the notion of a design pattern or data structure instance we implicitly add to each stereotype a special stereotype item *Elements* of type *Reg*. For an instance of a design pattern or data structure, instance *Elements* contains the set of objects which participates in the instance. Later on we will show how this property can be formalized as part of the stereotype system invariant.

Definition 2 (Stereotype item *Elements*)**.** *Elements is a stereotype item of type* Reg *which is implicitly added to all stereotypes.*

As we have mentioned above we would like to use the same stereotype to specify various parts of a program. It is possible that the same part of the code participates in different instances of a stereotype. For instance, the same class can simultaneously play the role of a subject in the observer design pattern and the role of a list in the iterator design pattern. To be able to use the same stereotype twice in the same part of code we introduce stereotype slices. A stereotype slice of type *St* is an instantiation of a stereotype *St*.

Definition 3 (Stereotype slice)**.** *A stereotype slice of type* St *is a polymorphic partial map from a pair* $\langle o, \mathrm{StItem}\rangle$ *into a value of type* T*, where* o *is* *a reference and* StItem *is a stereotype item of type T of stereotype St. We denote a stereotype slice as* StSlice*. We denote a value of a stereotype slice* StSlice *on a pair* $\langle o, \text{StItem}\rangle$ *as* o.StSlice.StItem*. If it is clear from the context which stereotype slice we imply in the expression* o.StSlice.StItem *then the expression can be abbreviated as* o.StItem*.*

You can notice that the above definition of stereotype slice is very similar to the heap definition provided by some verification approaches (e.g., Spec# [7]). The reason for this is that an instantiation of a stereotype item is a ghost field. Such a field is used for specification and verification but dropped by a compiler and not visible during a program execution. Another way to formalize the instantiation of stereotype items is to add them as ghost fields to the heap. For technical reasons we prefer to formalize a stereotype slice as an entity separate from the heap. In this way it is simpler to deal with multiple stereotype slices of the same stereotype and specify properties of stereotype operations which we consider below.

Let us consider an example of a stereotype instance. Let us assume that there are three lists o'_1, o'_2, and o'_3 and five iterators o^1_1, o^2_1, o^3_1, o^1_2, and o^2_2. o^1_1, o^2_1, and o^3_1 refer to o'_1; o^1_2 and o^2_2 refer to o'_2. A stereotype slice of type `RelationInversion` is described by the following table:

	sink	*Source*
o'_1	o'_1	$\{o^1_1, o^2_1, o^3_1\}$
o^1_1	o'_1	$\{o^1_1, o^2_1, o^3_1\}$
o^2_1	o'_1	$\{o^1_1, o^2_1, o^3_1\}$
o^3_1	o'_1	$\{o^1_1, o^2_1, o^3_1\}$
o'_2	o'_2	$\{o^1_2, o^2_2\}$
o^1_2	o'_2	$\{o^1_2, o^2_2\}$
o^2_2	o'_2	$\{o^1_2, o^2_2\}$
o'_3	o'_3	$\varnothing$

In the above table a row corresponds to an object and a column corresponds to a stereotype item. For instance, the second row represents the values of the stereotype items of object o'_1 and the second column represents the values of the stereotype item *sink* for various objects. The intersection of a row which corresponds to an object o and of a column which corresponds to a stereotype item *StItem* is equal to *o.StItem*. For instance, sine there is o'_1 on intersection of the row o'_1 and of the column *sink* we can conclude that $o'_1.sink = o'_1$.

A visualization of the above stereotype slice is shown on Figure 3.5. Black circles denote objects and arrows denote function f from the previous section. Objects in the slice are split into disjoint groups. On Figure 3.5 objects which belong to the same group are bounded by a big circle. The interior of the circle contains an object which corresponds to the stereotype item *sink* and a set of objects which corresponds to the *Source*. We call this group of disjoint objects *stereotype instance*. The formal definition of *stereotype instance* is provided in the next section.

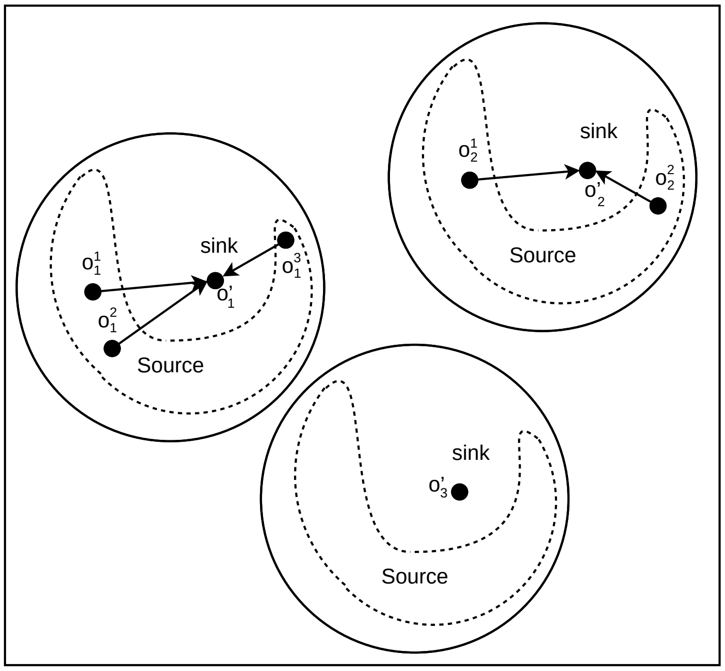


Figure 3.5: A visualization of a stereotype slice of type `RelationInversion`

We can see that there is a lot of redundancy in the above stereotype slice. The same values of the stereotypes *sink* and *Source* are duplicated by each member of a stereotype instance. First of all we would like to note that this duplication does not occur in general. For most stereotype items the values are different for different objects. For instance, in **Chapter 4** we consider a "sequence" stereotype which can be used to specify a linked list. The "sequence" stereotype has a stereotype item which contains a reference to the next element of the sequence. The value of this stereotype item differs for each object.

Stereotype items and slices describe which ghost fields are needed to specify the desired properties. Nevertheless, they do not describe these properties. For instance, we have mentioned above that the *Elements* stereotype item captures elements of a stereotype instance but have not described this property explicitly. To formalize and reuse properties of stereotype items we introduce stereotype invariants which are considered in the next subsection.

3.2.2 Stereotype invariants

A stereotype invariant is a property of stereotype items. In our approach we restrict ourself to universally quantified properties, because they are more feasible for automatic verification with **SMT** theorem provers. A stereotype invariant is used to construct the stereotype system invariant. We call it system invariant because later on we will prove that it holds at every point of verified program. The stereotype system invariant consists of two parts: one is common for all stereotypes and the other one is constructed from the stereotype invariant. The common part guarantees disjointness of stereotype instances. The rest of the subsection is organized as follows. First we describe the common part of the system invariant, then provide a definition of stereotype invariants and describe how they can be instantiated to construct the stereotype system invariant.

To formalize the stereotype system invariant we have to introduce one more notion: participation of an object in a stereotype slice. This notion is based on the participation of a class in a stereotype slice which will be introduced in **Section 3.4**. Each class C explicitly lists stereotype slices $StSlice_1, \ldots, StSlice_n$ which are used to specify properties of class C. We say that class C participates in stereotype slices $StSlice_1, \ldots, StSlice_n$. We say that an object o participates in a stereotype slice *StSlice* if and only if the type of o is class C and C participates in *StSlice*. To formalize this aspect of stereotype slices we introduce function **Dom** which for a given stereotype slice provides a set of object references which participate in the stereotype slice. A description of how **Dom** changes during program execution and the relation between values of **Dom** and object types is provided in **Section 3.3** and **Section 3.4**. For now, we just assume the existence of **Dom** and use it in invariant definitions.

Definition 4 (Function **Dom**)**.** *Function **Dom** is a map from a stereotype slice to a set of references. We say that an object o participates in a stereotype slice* StSlice *if and only if* $o \in \textbf{\textit{Dom}}(\text{StSlice})$.

We begin the description of the stereotype system invariant with the specification of properties of the *Elements* stereotype item. *Elements* contains exactly the objects of a stereotype instance.

Definition 5 (*Elements* system invariant)**.** *We say that the system invariant of* Elements *holds for a stereotype slice* StSlice *(denoted by* ***SysInvEl***[StSlice]*) if:*

- $\forall o, o' \in \textbf{\textit{Dom}}(\text{StSlice}) : o'.\text{StSlice.Elements} = o.\text{StSlice.Elements} \lor o'.\text{StSlice.Elements} \sharp o.\text{StSlice.Elements}$.
  This invariant guarantees consistency of the Elements *stereotype item's values. Values of the* Elements *stereotype item are either equal or disjoint, but never partially intersect.*

- $\forall o \in \boldsymbol{Dom}(\mathrm{StSlice}) : o \in o.\mathrm{StSlice.Elements}$.
  This invariant guarantees that an objet participates in its own stereotype instance.

  The rest of invariants guarantees that a value of a stereotype item does not go beyond the stereotype instance.

- *For each stereotype item* StItem *of type* Reg *the following holds:*
  $\forall o \in \boldsymbol{Dom}(\mathrm{StSlice}) : o.\mathrm{StSlice.StItem} \subseteq o.\mathrm{StSlice.Elements}$.

- *For each stereotype item* StItem *of type* ref *the following holds:*
  $\forall o \in \boldsymbol{Dom}(\mathrm{StSlice}) : o.\mathrm{StSlice.StItem} \in o.\mathrm{StSlice.Elements}$.

- *For each stereotype item* StItem *of type* ref $\rightarrow$ Reg *the following holds:*
  $\forall o, o' \in \boldsymbol{Dom}(\mathrm{StSlice})$:
  $o.\mathrm{StSlice.StItem}[o'] \subseteq o.\mathrm{StSlice.Elements}$.

- *For each stereotype item* StItem *of type* $\mathrm{ref}^2 \rightarrow$ Reg *the following holds:*
  $\forall o, o', o'' \in \boldsymbol{Dom}(\mathrm{StSlice})$:
  $o.\mathrm{StSlice.StItem}[o', o''] \subseteq o.\mathrm{StSlice.Elements}$.

- *For each stereotype item* StItem *of type* ref $\rightarrow$ ref *the following holds:*
  $\forall o, o' \in \boldsymbol{Dom}(\mathrm{StSlice})$:
  $o.\mathrm{StSlice.StItem}[o'] \in o.\mathrm{StSlice.Elements}$.

- *For each stereotype item* StItem *of type* $\mathrm{ref}^2 \rightarrow$ ref *the following holds:*
  $\forall o, o', o'' \in \boldsymbol{Dom}(\mathrm{StSlice})$:
  $o.\mathrm{StSlice.StItem}[o', o''] \in o.\mathrm{StSlice.Elements}$.

The first two invariants guarantee that values of *Elements* form a partition [20] of the set of objects which participate in the stereotype slice *StSlice*. Namely, the values of *Elements* cover all objects which participates in the stereotype slice *StSlice* and they are pairwise disjoint. We can reformulate this property as the following one: each object participates in exactly one stereotype instance.

It is typical that a block of the partition which is defined by values of *Elements* corresponds to an instance of a design pattern or a data structure. Therefore, we call such a block of the partition stereotype instance.

Definition 6 (Stereotype instance of a stereotype slice *StSlice*)**.** *We call subset of* $\boldsymbol{Dom}(\mathrm{StSlice})$ StInst *stereotype instance if and only if* $\forall o \in \mathrm{StInst} : o.\mathrm{StSlice.Elements} = \mathrm{StInst}$*. We say that an object* o *participates in a stereotype instance* StInst *if and only if* $o \in \mathrm{StInst}$.

If **SysInvEl**[*StSlice*] holds then a stereotype slice is a disjoint union of stereotype instances of *StSlice*. An update of a stereotype instance does not affect other stereotype instances. In **Chapter 3.3** we rely on this property of stereotype slices when we describe stereotype operations.

Up to now we identify a stereotype instance *StInst* by the set of elements which participates in *StInst*. Therefore in order to check equality or inequality of a pair of stereotype instances we have to check equality or inequality of the corresponding sets. Since this operation is widely used in our approach we would like to simplify it. To achieve it we introduce a unique identifer for each stereotype instance: an expression of reference type which depends on stereotype items and uniquely identifies a stereotype instance. We refer to the instance identifer as `instID`. `instID` has to be provided by a user as part of the stereotype declaration.

Definition 7 (`instID`)**.** *`instID` is a user provided expression which depends on a free variable o and the values of stereotype items.*

For instance, for the `RelationInversion` stereotype we define `instID` in the following way: `instID`=`if`(o.*sink*= **null**) `then` o `else` $o.sink$. Here we use the logical statement "if" to formalize `instID`. The definition considers two cases: either the sink of an instance in which object o participates is equal to null. According to the user provided stereotype invariants which we consider below, this is the case when the stereotype instance is a singleton and contains only the object o. Therefore in this case we use o as `instID`. In the second case $o.sink$ is not null, and therefore **SysInvEl** implies that $o.sink$ has to be different for different instances.

We usually use `instID` to identify for a given object a stereotype instance to which the object belongs. Therefore we introduce a special notation to check this property. For an expression *exp* of a reference type *exp.StSlice*.`instID` denotes the result of the substitution of *exp* for o in the expression which defines `instID` for *StSlice*. The stereotype slice can be omitted if it is clear from the context. In this case we write just *exp*.`instID`. For instance, for a variable v of a reference type and a stereotype slice *StSlice* of type `RelationInversion` we interpret $v.StSlice.instID$ as `if`(v.*sink*= **null**) `then` v `else` $v.sink$.

The following invariant formalizes the desired properties of `instID`.

Definition 8 (`instID` system invariant)**.** *We say that the system invariant of `instID` holds for a stereotype slice* StSlice *(denoted by* ***SysInvID***[StSlice]*) if:*

- $\forall o \in \boldsymbol{Dom}(\text{StSlice}) : o.\text{StSlice}.\mathtt{instID} \in o.\text{StSlice.Elements}$
  An instance ID is an element of the instance.
- $\forall o, o' \in \boldsymbol{Dom}(\text{StSlice}) : o \in o'.\text{StSlice.Elements} \Leftrightarrow o.\text{StSlice}.\mathtt{instID} = o'.\text{StSlice}.\mathtt{instID}$.
  A pair of objects belongs to the same stereotype instance if and only if their `instID`s are equal.

A proof obligation which checks the uniqueness property of `instID` has to be verified for each stereotype. We formalize this proof obligation in the next subsection.

By this we conclude the consideration of the common part of the stereotype system invariant and move on to the consideration of the stereotype specific part of the stereotype system invariant.

Definition 9 (Stereotype invariant)**.** *A stereotype invariant of a stereotype* St *is a user-provided formula which depends on stereotype items of stereotype* St*. It is universally quantified over variables of reference types.*

As we have mentioned above we would like to preserve the disjointness of stereotype instances. In this way we can be sure that an update of a stereotype instance does not affect any other instance. To achieved disjointness we need not only element disjointness but also to be sure that an update of a stereotype instance does not violate an invariant of another instance. Therefore we restrict each invariant to elements of a stereotype instance. We achieve it in the following way. To each stereotype invariant we add a quantifier over all stereotype instances and restrict the range of the remaining universal quantifiers to elements of the stereotype instance. The above definition is formalized as an instantiation of stereotype invariants below. This simplifies the verification of stereotype properties significantly. Since most of the time we are interested only in objects which share the same instance of a design pattern or a data structure, even with this constraint stereotype invariants are flexible enough.

Let us consider the stereotype invariants of `RelationInversion`. Please note that the in the stereotype definitions the range of the quantification is not specified. The range of the quantification is defined during the instantiation of the stereotype invariants which we consider a bit later.

- $\forall o : o.sink \notin o.Source$
  The first invariant states that an object cannot depend on itself. For instance, in the mediator design pattern an object cannot simultaneously play the role of the mediator and of a colleague in the same instance of the design paten. This invariant also guarantees acyclicity of the function f from the previous section.

- $\forall o : o.sink = null \Rightarrow o.Source = \{o\}$
  The second invariant guarantees the absence of redundant semantic relations. Let us explain it on the following example. Let us assume that there are two observers which are not attached to a specific subject. There is no dependence between them, therefore the update of one of them should not affect the other one. Therefore it makes sense to keep them in disjoint stereotype instances. In this way the independence between the observers will be guaranteed by the stereotype system invariant. Without the above invariant these two observers could belong

to the same stereotype instance. In that case the disjointness cannot be verified so easily. Therefore the above invariant states that if an object is not attached to a source it has to form a singleton stereotype instance.

- $\forall o : o.Elements = o.Source \cup \{o.sink\}$
  The third invariant specifies which objects participate in a stereotype instance. An object participates in an instance if and only if it is either equal to *sink* or belongs to *Source*.

  The last two invariants have a more technical nature. They state that the values of *Source* and *sink* are equal for all objects from the same stereotype instance. We need these two invariants to guarantee consistency between copies of *Source* and *sink* that are stored in different objects.

- $\forall o, o' : o'.Source = o.Source$
- $\forall o, o' : o'.sink = o.sink$

Now let us define the instantiation of stereotype invariants for a given stereotype slice. We begin with the definition of a formula instantiation for a given stereotype slice.

Definition 10 (Instantiation of a formula for a stereotype slice)**.** *For an arbitrary formula φ which depends on stereotype items of a stereotype* St *and for an arbitrary stereotype slice* StSlice *of type* St *we construct an instantiation of φ for* StSlice *by replacing all instances of stereotype items* StItem *of stereotype* St *by* StSlice.StItem*. We denote the resulting formula by φ*[StInst]*.*

Using the instantiation of a formula for a stereotype slice we can define the instantiation of stereotype invariants on a stereotype instance and slice. For simplify, in the below definition we consider the instantiation of only one invariant. The generalization in case of several invariants is straightforward and therefore omitted.

Definition 11 (Stereotype instance invariant)**.** *We denote a stereotype invariant of a stereotype instance* StInst *of a stereotype slice* StSlice *of a stereotype* St *as $\boldsymbol{Inv}_{\mathrm{St}}[\mathrm{StSlice}, El]$, where El is the set of elements of* StInst*. If a stereotype invariant is $\forall o_1, \ldots, \forall o_n : \varphi[\mathrm{StSlice}]$ then $\boldsymbol{Inv}_{\mathrm{St}}[\mathrm{StSlice}, El] = \forall o_1 \in El, \ldots, \forall o_n \in El : \varphi[\mathrm{StSlice}]$.*

The above definition defines the range of the quantifiers as the elements of the stereotype instance for which the invariant is instantiated.

We call a stereotype instance with elements El consistent if
$\mathbf{Inv}_{St}[StSlice, El]$ holds.

Definition 12 (Stereotype slice invariant)**.** *We denote the stereotype invariant of a stereotype slice* StSlice *of a stereotype* St *as*
$\boldsymbol{Inv}_{\mathrm{St}}[\mathrm{StSlice}] = \forall\ id \in \boldsymbol{Dom}(\mathrm{StSlice}) : \boldsymbol{Inv}_{\mathrm{St}}[\mathrm{StSlice}, id.\mathrm{Elements}]$.

The above definition quantifies over all objects which participate in the stereotype slice *StSlice* to access all stereotype instances of *StSlice*. *id.Elements* contains objects which participate in the same stereotype instance.

Let us consider an example of a stereotype instantiation. Let us assume that *StSlice* is a stereotype slice of type `RelationInversion`. An instantiation of the first stereotype invariant for a stereotype slice *StSlice* is the following: $\forall\ id \in \mathbf{Dom}(StSlice), o \in id.Elements : o.sink \notin o.Source$.

Now we combine all of the above invariants definitions to define the stereotype system invariant. In **Section 3.3** we prove that the stereotype system invariant holds at every program point.

Definition 13 (Stereotype system invariant for a stereotype slice *StSlice*)**.** *We say that the stereotype system invariant holds for a stereotype slice* StSlice *if* $\boldsymbol{Inv}_{\mathrm{St}}[\mathrm{StSlice}]$*,* $\boldsymbol{SysInvEl}[\mathrm{StSlice}]$*, and* $\boldsymbol{SysInvID}[\mathrm{StSlice}]$ *hold. We denote the stereotype system invariant for a slice* StSlice *as*
$\boldsymbol{SysInv}[\mathrm{StSlice}]$*.*

We call a stereotype slice *StSlice* consistent if $\mathbf{SysInv}[StSlice]$ holds.

We have seen that quite often a stereotype item has the same value for all objects participating in a stereotype instance. For instance, in the `RelationInversion` stereotype both stereotype items have this property. We call such stereotype items static and handle them in a special way. By this we achieve more concise specifications.

Definition 14 (Static stereotype item)**.** *We call a stereotype item static if its definition in the stereotype definition begins with the keyword* `static`*.*

We capture properties of static stereotype items by the following invariants.

Definition 15 (Implicit invariant for a static stereotype item)**.** *For each static stereotype item* StItem *of stereotype* St *we add an implicit invariant* $\forall o, o' : o'.\mathrm{StItem} = o.\mathrm{StItem}$ *to the definition of the stereotype* St*.*

Static stereotype items have one more advantage. Since the value of a stereotype item is the same for all objects which participate in a stereotype instance, we can omit the receiver when mentioning a static stereotype item in a stereotype slice. In the instantiation of an invariant we use *id* from **Definition 12** as a receiver of the static stereotype item.

Let us demonstrate the benefits of static stereotype items on the example of the `RelationInversion` stereotype. If we declare both *Source* and *sink* as static we can drop the last two invariants since they are added implicitly and simplify the remaining invariants in the following way:

- $sink \notin Source$
- $\forall o : sink = null \Rightarrow Source = \{o\}$
- $Elements = Source \cup \{sink\}$

To illustrate how the instantiation of an invariant which mentions a static field looks like let as consider the updated version of the first invariant. The instantiation of the updated version of the first stereotype invariant for a stereotype slice *StSlice* is
$\forall\ id \in \mathbf{Dom}(StSlice) : sink \notin Source.$

3.2.3 Stereotypes

In this subsection we put the definitions from the above subsections together and define stereotypes.

Definition 16 (Stereotype)**.** *A stereotype definition consists of:*

- *Stereotype header, which begins with the keyword* ***`Stereotype`*** *after which a name of the defined stereotype follows.*
- *Stereotype item definition, which begins with the keyword* ***`items`*** *after which a list of semicolon separated stereotype item definitions follows.*
- *Stereotype invariants definition, which begins with the keyword* ***`invariants`*** *after which a list of stereotype invariants follows.*
- *Instance identifer definition, which begins with* ***`instID`*** $=$ *after which the definition of* ***`instID`*** *follows.*

Let us consider an example of a stereotype definition. On Figure 3.6 we present the definition of the `RelationInversion` stereotype. All parts of the definition were already considered in the corresponding subsections.

As we have mentioned above for each stereotype we have to check that the definition of `instID` is a unique stereotype instance identifier. To validate this property the following proof obligation has to be verified for each stereotype.

Definition 17 (Stereotype proof obligation)**.** *For each stereotype* St *the following proof obligation has to be verified* $\forall$ StSlice : $\boldsymbol{SysInvEl}$[StSlice] $\wedge$ $\boldsymbol{Inv}_{\mathrm{St}}$[StSlice] $\Rightarrow$ $\boldsymbol{SysInvID}$[StSlice]*. Here the range of the quantification is stereotype slices of type* St*.*

The above proof obligation checks that if the *Elements* invariant and the stereotype invariants hold for a stereotype instance then for the same stereotype instance the `instID` invariant holds.

```
Stereotype RelationInversion{
items
    static $Source$: $Reg$ ;
    static $sink$: $ref$ ;
invariants
    $sink \notin Source$
    $\forall o : sink = \mathbf{null} \Rightarrow Source = \{o\}$
    $Elements = Source \cup \{sink\}$
instID = if($o.sink = \mathbf{null}$) then o else o.$sink$
}
```

Figure 3.6: The relation inversion stereotype.

3.3 Stereotype operations

In this section we consider stereotype operations. Stereotype operations construction is one of the most challenging parts of our stereotype-based approach. The complete description of stereotype construction methodology includes a lot of technical details. Therefore we only provide an informal description and examples of various stereotype operations here. The complete definitions and proofs are provided in **Chapter 6**.

Stereotype operations describe reusable transformations of stereotype slices. As we have mentioned above stereotypes facilitate reusability of specifications that describe a single program point. In a similar way stereotype operations facilitate reusability of specifications which establish a relation between different states.

Stereotype operations have two applications in our methodology. First they are used to specify how stereotype slices change during the program execution. To specify an update of a stereotype slice the user has to add a call of a stereotype operation to the program. Another application of stereotype operations is the specification of class methods and loop invariants. We consider both of these applications in **Section 3.4**.

We are interested not in arbitrary transformations but only in transformations which satisfy some properties. First of all we expect that stereotype operations preserve the stereotype system invariant which is defined in the previous section. Another property which we expect from stereotype operations is consistency. The consistency property guarantees that calls to stereotype operation never produce false assumptions. A violation of this property can affect soundness of program verification. We formalize these properties in **Chapter 6**.

We begin the construction of operations for a given stereotype with the basic operations. Basic operations are used to specify the simplest transformations of stereotype slices. For them we explicitly check the preservation of the stereotype system invariant and consistency. Since most stereotypes are approximations of binary relations, the basic operations for these stereotypes are addition and removal of the relation between two objects. For instance, in this section we consider the basic operations for `RelationInversion`: addition and removal of a relation between a source and a sink. In **Chapter 4** we specify basic operations for tree and sequence stereotypes. The basic operations of the tree stereotype are the addition and the removal of a subtree. The basic operations of the sequence stereotype are the concatenation of two sequences and splitting a sequence.

We have a special kind of basic operation which is responsible for the initialization of stereotype items for freshly allocated objects. This operations are called stereotype constructors. A stereotype constructor executed for a stereotype slice *StSlice* adds an initialized object to **Dom**(*StSlice*) which is defined in the previous section. In this section we describe two stereotype constructors for `RelationInversion`. One is used to initialize a source and the other one is used to initialize a sink. Constructors of the tree and sequence stereotypes create a singleton tree and sequence, respectively.

We consider the basic operations of `RelationInversion` in **Subsection 3.3.1**.

Basic stereotype operations can be used to construct more complicated stereotype operations. For this purposes we introduce a special language for combining stereotype operations. We call the language specification language of specifications or **SLS**. **SLS** is the language to specify stereotype operations which are then used to specify the real source code. We briefly describe **SLS** in **Subsection 3.3.2**. The complete definitions and proofs are provided in **Chapter 6**.

Since a stereotype operation is a reusable transformation specification, the more predefined operations are provided to a user the less work has to be done by the user. Therefore it is strongly desirable to construct for a stereotype a stereotype operation which can be used to define any other transformation. We call such stereotype operations *universal transformations*. If a user-provided stereotype is equipped with a universal transformation then a stereotype user does not need to spend efforts on constructing stereotype operations. Instead a universal transformation can be instantiated to achieve a desired stereotype operation. Since the construction of the stereotype operations is the most challenging part of the stereotype methodology, a predefined universal transformation for a stereotype simplifies the usage of stereotypes significantly. We discuss universal transformations in more details at the end of **Chapter 6**. We provide specifications of universal transformations for sequence and tree stereotypes in **Appendix A** and **Appendix B**, respectively.

3.3.1 Basic stereotype operations

In this subsection we consider basic operations and constructors on the example of `RelationInversion`. The basic stereotype operations of `RelationInversion` are shown on Figure 3.7. `addRelationInversion` adds a relation between a source an a sink, and `removeRelationInversion` removes a relation between a source and a sink.

```
addRelationInversion⟨RelationInversion⟩($source : ref!, sink : ref!$)
{
Pre-conditions:
    $sink.sink = sink$ ;
    $source.sink = \mathbf{null}$ ;
Input instances:
    $inInst_1 = source$ ;
    $inInst_2 = sink$ ;
Output instances:
    $outInst = inInst_1 \cup inInst_2$ ;
Transformations:
    $\langle source, sink, sink \rangle$ ;
    $\langle outInst, Source, sink.Source \cup \{source\} \rangle$ ;
}

removeRelationInversion⟨RelationInversion⟩($source : ref!$)
{
Pre-conditions:
    $source.sink \neq \mathbf{null}$ ;
    $source.sink \neq source$ ;
Input instances:
    $inInst = source$ ;
Output instances:
    $outInst_1 = \{source\}$ ;
    $outInst_2 = inInst \setminus \{source\}$ ;
Transformations:
    $\langle source, sink, \mathbf{null} \rangle$ ;
    $\langle source, Source, \{source\} \rangle$ ;
    $\langle outInst_2, Source, source.Source \setminus \{source\} \rangle$ ;
}
```

Figure 3.7: The `addRelationInversion` and `removeRelationInversion` stereotype operations of the `RelationInversion` stereotype

Let us first consider `addRelationInversion`. The header of the operation begins with the name of the operation `addRelationInversion`. Then the list of the affected stereotype slices follows. The operation affects one stereotype slice of type `RelationInversion`. If an operation affects more than one stereotype slice of the same type then the names of stereotype slices have to be provided for each affected stereotype slice. Otherwise the stereotype name is used as a name of the stereotype slice. For instance, we use the name of stereotype `RelationInversion` to name the only stereotype slice affected by `addRelationInversion`.

Most of the stereotype operations affect only one stereotype slice. Nevertheless, in some cases an operation can use the values of the stereotype items of one stereotype slice to update the value of another stereotype slice. For instance, in **Section 5.4** we introduce a stereotype operation `acquireBlocked`. The operation plays a key role in the specification and verification of the implementation of **PIP** which we discussed in **Chapter 2**. `acquireBlocked` uses information about a tree path which is provided by a tree stereotype slice to create a stereotype instance which represents this path in a sequence stereotype slice.

The header is completed by the list of input parameters. `addRelationInversion` has two input parameters; *source* are *sink* are references to a source and a sink of the added relation in the `RelationInversion` stereotype slice.

Quite often stereotype operations have extra requirements for input parameters of reference an region types regarding the **null** reference. Quite often it is expected that the value of an input parameter of reference type is not null and the value of an input parameter of region type does not contain **null**. These properties can be captured by corresponding pre-conditions. To abbreviate the specification we introduce special notations which implicitly add the corresponding pre-condition to the stereotype operation. For each input parameter v whose type is defined as *ref!* we implicitly add to the operation definition the pre-condition $v \neq \mathbf{null}$ and replace the type of v by *ref*. In a similar way for each input parameter R whose type is defined as *Reg!* we implicitly add to the operation definition the pre-condition $\mathbf{null} \notin R$ and replace the type of R by *Reg*. The notation is inspired by Spec# non-null types [49]. Since the types of both *source* and *sink* are defined as *ref!* we implicitly add to the `addRelationInversion` definition the pre-conditions $source \neq \mathbf{null}$ and $sink \neq \mathbf{null}$.

The body of the operation begins with the pre-conditions. A pre-condition describes the state of the stereotype slice and input parameters before the operation execution. The state of the stereotype slice is described as properties of the values of stereotype items. For each stereotype operation implicitly assumes the system invariants of all affected stereotype slices.

Let us recall that according to **Definition 3** we denote the value of stereotype sice *StSlice* on a pair $\langle o, StItem \rangle$ as $o.StSlice.StItem$. If the stereotype slice is clear from the context then we denote it as $o.StItem$. `addRelationInversion` affects only one stereotype slice which has the same name as its stereotype, `RelationInversion`. Since `addRelationInversion` affects only one stereotype slice we can abbreviate a stereotype item access $v.\texttt{RelationInversion}.StItem$ as $v.StItem$.

The first pre-condition states that the input parameter *sink* plays the role of the sink in the stereotype instance to which it belongs. The second pre-condition specifies that *source* is not attached to a sink. This pre-condition and the stereotype system invariant imply that the stereotype instance in which *source* participates is a singleton stereotype item which consists only of *source*.

The next part of the operation's body is input instances. Here we specify which instances can be affected by the operation. We use this information to define the operation. The operation preserves the values of all stereotype items which do not belong to the input instances. The input instances are identified by their instance identifier. An instance identifier can be extracted from each object which participates in the instance with the help of `instID`.

An input stereotype instance definition has the following syntax $InstanceName = instanceElement$, where $InstanceName$ is used in the operation definition to identify the instance and *instanceElement* is an object which participates in the instance. The instance identifier of the instance $InstanceName$ is $instanceElement.\texttt{instID}$. For instance, `addRelationInversion` has two input instances $inInst_1$ and $inInst_2$ whose instance identifers are $source.\texttt{instID}$ and $sink.\texttt{instID}$, respectively.

In a stereotype operation definition we use an input instance name to denote the set of objects which belongs to the stereotype instance. For instance, in the definition of `addRelationInversion` we use $inInst_1$ and $inInst_2$ to denote $source.Elements$ and $sink.Elements$, respectively.

As we have mentioned above we use input instances to define a frame of a stereotype operation. Since `addRelationInversion` has two input instances its frame can be define as $inInst_1 \cup inInst_2$, which is equivalent to $source.\texttt{instID} \cup sink.\texttt{instID}$. `addRelationInversion` does not affect values of stereotype items of objects which do not belong to $inInst_1 \cup inInst_2$.

A stereotype operation execution can transfer an object from one stereotype instance to another. We describe such transformations of stereotype instances in the "`Output instances:`" section of a stereotype operation definition. This part of the operation definition lists the output instances.

An output instance definition has the following syntax $InstanceName = Elements$, where $InstanceName$ is used in the operation definition to identify the instance and *Elements* is a set of objects which participate in the output instance. For instance, `addRelationInversion` has one output instance *outInst* which consists of the objects in $inInst_1 \cup inInst_2$. In other words `addRelationInversion` merges two input instances into one output instance.

For each stereotype operation we check that the set of elements of the input instances is equal to the set of objects in the output instances. The corresponding proof obligation is generated and checked for each stereotype operation. For instance, for the stereotype operation `addRelationInversion` we check the proof obligation $outInst = inInst_1 \cup inInst_2$. The validity of the proof obligation trivially follows from the definition of $outInst$.

The last part of the operation is `Transformations:`. It describes how the operation execution changes the value of the stereotype items of the input stereotype instances. The transformation is described as a semicolon separated sequence of transformation rules. Each transformation rule is a triple $\langle S, \mathit{StItem}, exp\rangle$ where S is a set of objects, StItem is a stereotype item of type T and exp is an expression of type T. The semantics of $\langle S, \mathit{StItem}, exp\rangle$ is defined in the following way $\forall \mathbf{v} \in S \Rightarrow \mathit{StSlice}'.\mathbf{v}.\mathit{StItem} = exp$, where $\mathit{StSlice}'$ is the value of the stereotype slice after the operation execution. In other words for all objects from S, the value of the stereotype item StItem of the stereotype slice after the program execution is equal to exp. The description of S and exp can depend on the value of the stereotype slice before the operation execution. exp also can depend on the value of variable $\mathbf{v}$ which is used for iteration over the objects in S. If the set of updated objects is a singleton set $S = \{o\}$ then we instead of $\langle S, \mathit{StItem}, exp\rangle$ write $\langle o, \mathit{StItem}, exp\rangle$.

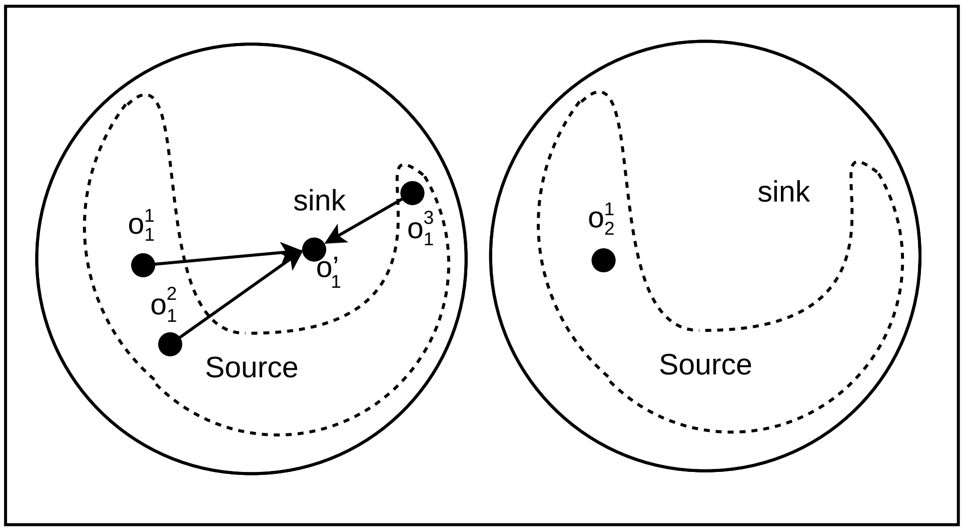
In the definition `addRelationInversion`, the transformation consists of two transformation rules. The first one is $\langle source, sink, sink\rangle$. This rule sets the value of stereotype item $sink$ of object $source$ to the object referenced by the input parameter $sink$. The second transformation rule $\langle outInst, Source, sink.Source \cup \{source\}\rangle$ sets the values of stereotype item $Source$ of all objects from both input instances into $sink.Source \cup \{source\}$.

If a value of a stereotype item of an object from an input instance is not mentioned by the transformation rules of the stereotype operation then the value is preserved by the stereotype operation. For instance, the transformation rules of `addRelationInversion` say nothing about the values of stereotype item $sink$ of objects from $inInst_2$. Therefore these values are preserved by the operation.

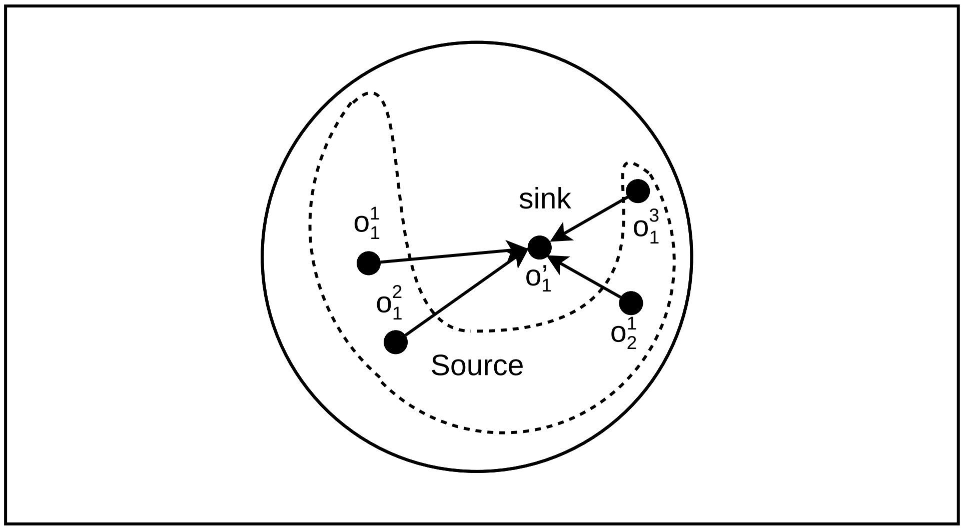
Transformation rules in a definition of a stereotype operation are possibly inconsistent. For instance, if the dedition of a stereotype operation includes transformation rules $\langle S_1, \mathit{StItem}, exp_1\rangle$ and $\langle S_2, \mathit{StItem}, exp_2\rangle$, and there is an object o such that $o \in S_1 \cap S_2$ and $exp_1 \neq exp_2$ then the description of the stereotype slice after the operation execution is inconsistent. One rule states that $o.\mathit{StSlice}'.\mathit{StInst} = exp_1$ and another states that $o.\mathit{StSlice}'.\mathit{StInst} = exp_2$, where $\mathit{StSlice}'$ is a value of stereotype slice after the operation execution. To prevent such an inconsistency we check the proof obligation $S_1 \sharp S_2$ for each pair of transformation rules $\langle S_1, \mathit{StItem}, exp_1\rangle$ and $\langle S_2, \mathit{StItem}, exp_2\rangle$ of each stereotype operation.

The basic operation has to preserve the stereotype system invariant. Therefore, for each stereotype operation we check the proof obligation that the output stereotype instance preserve the stereotype system invariant.

Since the values of elements which do not belong to the output instances are not affected by the operation the above proof obligation is strong enough to ensure preservation of the stereotype system invariant. For instance, for the operation `removeRelationInversion` we check the proof obligations $\mathbf{Inv}_{\texttt{RelationInversion}}[\texttt{RelationInversion}, outInst_1]$ and $\mathbf{Inv}_{\texttt{RelationInversion}}[\texttt{RelationInversion}, outInst_2]$.



(a) State of the stereotype slice *StSlice* before the stereotype operation execution.



(b) State of the stereotype slice *StSlice* after the stereotype operation execution.

Figure 3.8: The figure depicts an execution of the operation $\texttt{addRelationInversion}\langle \mathit{StSlice}\rangle(source, sink')$ where variable *source* refers in the object o_2^1 and variable *sink* refers in object o_1'.

Let us consider an example of an execution of the `addRelationInversion` stereotype operation. The state of the stereotype slice *StSlice* is shown on Figure 3.8(a). Variable *source* refers to the object o_2^1 and variable *sink* refers to object o_1'. The state of *StSlice* after the execution of `addRelationInversion`$\langle$*StSlice*$\rangle$$(source, sink')$ is shown on Figure 3.8(b).

The second stereotype operation which is defined in Figure 3.7 is `removeRelationInversion`. `removeRelationInversion` is the inversion of `addRelationInversion`. It takes one input stereotype instance and splits it in two by moving an object from the source of the input instance into a new singleton stereotype instance.

The only input parameter of the operation *source* is a reference to the object which is moved by the operation in a disjoint stereotype instance. The pre-condition of the operation checks that the stereotype instance to which *source* belongs is not a singleton and *source* does not play the role of the sink in the stereotype. The only stereotype instance which is affected by an operation execution is $inInst$. The instance identifer of $inInst$ is *source*.`instID`. An operation execution splits $inInst$ in two disjoint stereotype instances $outInst_1$ and $outInst_2$. $outInst_1$ is a singleton stereotype instance which contains *source*. All other objects of $inInst$ belong to $outInst_2$. The first two transformation rules describe the values of *sink* and *Source* of the only object which belongs to $outInst_1$. The last transformation rule removes *source* from the value of stereotype item *Source* of stereotype instance $outInst_2$.

Let us consider an example of an execution of the `removeRelationInversion` stereotype operation. The state of the stereotype slice *StSlice* is shown on Figure 3.8(b). Variable *source* refers to the object o_2^1 and variable *sink* refers to object o_1'. The state of *StSlice* after the execution of `removeRelationInversion`$\langle$*StSlice*$\rangle$$(source, sink')$ is shown on Figure 3.8(a).

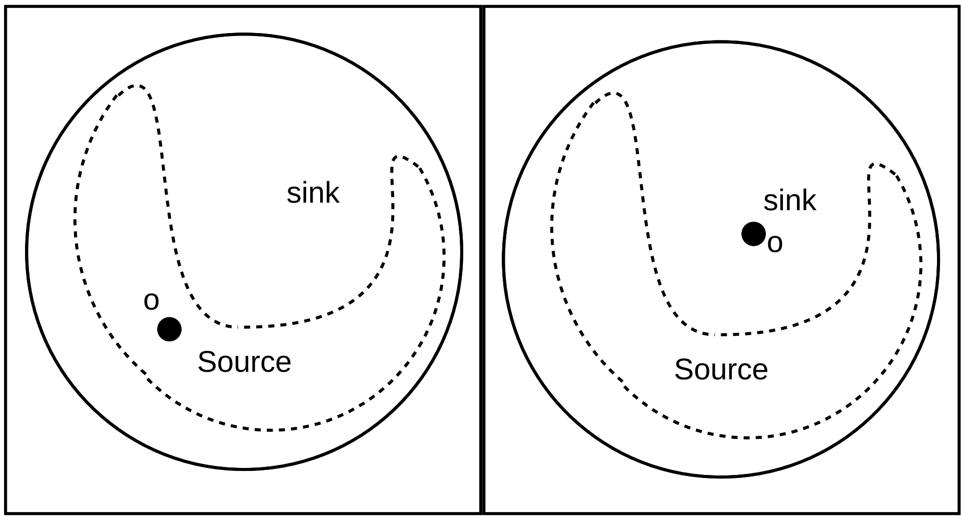
```
constructor createSource⟨RelationInversion⟩(source : ref)
{
    ⟨source, sink, null⟩ ;
    ⟨source, Source, {source}⟩ ;
}

constructor createSink⟨RelationInversion⟩(sink : ref)
{
    ⟨sink, sink, sink⟩ ;
    ⟨sink, Source, ∅⟩ ;
}
```

Figure 3.9: Constructors of the `RelationInversion` stereotype.

Let us know consider stereotype constructors. Stereotype constructors are a special kind of basic operation which is responsible for the initialization of the stereotype items of a freshly allocated object. A constructor always takes exactly one non-null input parameter of a reference type and it participates in exactly one stereotype slice. Since the only input parameter is a reference to a freshly allocated object there is no need for pre-conditions. We also assume that a constructor does not access the values of stereotype items of a freshly allocated object. The only output stereotype instance of a stereotype constructor is a singleton stereotype instance which contains the freshly allocated object. Since input and output instances are predefined for all constructors there are no input and output instances sections in a stereotype constructor definition. A constructor definition consists only of a transformation section which initializes the stereotype items of the singleton output instance.

In **Section 3.2.2** we introduced function **Dom** which yields the set of object references which participate in a stereotype slice. An execution of a stereotype constructor for stereotype slice *StSlice* and an object o adds o to **Dom**(*StSlice*).

For each stereotype constructor we check the proof obligation that the system invariant is established for the output singleton stereotype instance. Since the values of elements which do not belong to the output instance are not affected by the constructor this proof obligation is strong enough to ensure preservation of the stereotype system invariant.



(a) A singleton stereotype instance after an execution of `createSource`.

(b) A singleton stereotype instance after after an execution of `createSink`.

Figure 3.10: The stereotype instances after an execution of the constructors of the `RelationInversion` stereotype for a freshly allocated object o.

We define two constructors for `RelationInversion`on Figure 3.9. The first one is `createSource`. It is used to use a freshly allocated object as a source. The other one is `createSink`. It is used to use the object as a sink. To satisfy the stereotype invariants, `createSource` sets the sink of the input object to **null** and the source to the singleton set which contains the input object. `createSink` does the opposite. It sets the sink to the input object and source to the empty set.

The result of the execution of both stereotype constructors for a freshly allocated object o is shown in Figure 3.10. On the figures we show only stereotype instances in which o participates and ignore the other stereotype instances.

Quite often the same expressions over stereotype items can be used several times in a program specification. Since the expression size can be large we use functions to avoid duplications of such expressions.

We provide an example of a function definition on Figure 3.11. The function definition begins with the keyword `function`, followed by function type. The type of the function declared on Figure 3.11 is `Bool`. After the function type follows the function name, `isSingletonSource`. Similarly to the stereotype operation headers the function header is concluded by a declaration of the affected stereotype slices and input parameters. `isSingletonSource` affects a stereotype slice of type `RelationInversion` and has one input parameter o of type *ref*.

After the function header follows the function body. A function body is an expression of the function type. In case of the `isSingletonSource` function the body is the expression $o.sink = \mathbf{null}$ of type `Bool`. The function body must not be recursive. The only variables on which the function body depends are input parameters.

The function `isSingletonSource` checks if the input parameter belongs to a singleton stereotype slice and plays the role of the source. The function relies on the stereotype system invariant which states that if the sink of the stereotype is null then the source of the stereotype is a singleton set.

We treat functions as abbreviations. If an expression mentions a function then we refine the expression by replacing the function by the function body where input parameters are replaced by actual parameters. In this way all function applications can be eliminated from the expression. Therefore when we consider program verification we assume that there are no function applications.

If a function type is boolean then we call the function predicate. For instance, `isSingletonSource` is a predicate.

3.3.2 Specification language of specifications

In the previous subsection we considered basic stereotype operations. In this section we consider how to construct more advanced operations from the

```
function Bool isSingletonSource⟨RelationInversion⟩(o : ref)
    := o.sink = null
```

Figure 3.11: A function which checks if a given object belongs to a singleton relation inversion instance and play the role of the source.

basic operations. We call such operations composite stereotype operations. We introduce several operation composers which for given basic or composite operation produce a new composite operation. Here we briefly consider the specification language of specifications (denoted as **SLS**) which is used to construct composite operations.

A definition of a composite operation consists of a header and a body. The header of a composite operation has the same structure as the header of a basic operation. It defines input parameters, affected stereotype slices, and the operation name. The body of a composite stereotype operation is an **SLS** term which describes the transformation of the affected stereotype slices.

The syntax of **SLS** is shown on Figure 3.12. **SLS** consists of the following expressions: operation call, conditional statement, sequential composition, parallel composition, and skip operation. Since we use them to compose a more complicated stereotype operation from the simpler stereotype operations we also call them operation composers. The complete definition of the operational semantics of **SLS** and other relevant definitions and proofs are provided in **Section 6.3**.

$$\begin{array}{lll} \mathbf{t} & ::= & \mathbf{op}(exp_1, \ldots, \mathrm{exp}_n) \\ & | & \texttt{if } (\varphi)\ \mathbf{t}\ \texttt{else}\ \ \mathbf{t} \\ & | & \mathbf{t}; \mathbf{t} \\ & | & \overset{f_{\mathbf{Inst}}}{\underset{i \in Ind}{\parallel}} \mathbf{op}(exp_1[i], \ldots, \mathrm{exp}_n[i]) \\ & | & \texttt{skip} \end{array}$$

Figure 3.12: **SLS** syntax.

Below we provide an informal description of the operational semantics of **SLS**. Since stereotype operations describe transformations of the stereotype slices they can be naturally described in operational terms. Therefore the operational semantics of the most **SLS** expressions is similar to the operational semantics of the corresponding expressions of an imperative programming language. On the other hand in **Section 3.4** we describe how we use stereotype operations for real source code specifications. We cannot directly use operational descriptions of stereotype operations in specifications. Therefore in **Section 6.4** we describe how stereotype operations can be translated into logical descriptions in the universally quantified fragment of **FOL**.

A composite stereotype operation can include calls of both basic and composite operations. An operation call is denoted as $\mathbf{op}(\mathrm{exp}_1, \ldots, \mathrm{exp}_n)$ where **op** is the stereotype operation name and $\mathrm{exp}_1, \ldots, \mathrm{exp}_n$ are the actual parameters of the call. A basic operation call is a basic building block of the composite operations. A composite operation can call another composite operation. Recursive calls are treated in a special way. For each recursive operation the user has to provide extra specifications which are similar to a loop invariant and a measure which guarantees termination. In this subsection we do not consider recursive operations in detail. Several examples of recursive operations can be found in **Appendix A** and **Appendix B**.

A conditional statement is denoted as `if` (φ) **t** `else` **t** where φ is a condition. Sequential composition is denoted as **t**; **t**. The semantics of both is standard. If the condition of a conditional statement holds then the first term is executed, otherwise the second one. Quite often there is no need for an "else" statement therefore we abbreviate `if` (φ) **t** `else skip` as `if` (φ) **t**. A sequential composition transforms an input slice into an output slice if and only if there exists an intermediate stereotype slice such that the first term transforms the input slice into the intermediate slice and the second term transforms the intermediate slice into the output slice. The intermediate stereotype slice may not exist because of the pre-conditions violation.

The next operation composer is parallel composition, which is denoted as $\overset{f_{\mathbf{Inst}}}{\underset{i \in Ind}{\|}} \mathbf{op}(\mathrm{exp}_1[i], \ldots, \mathrm{exp}_n[i])$. Here **op** is the name of the stereotype operation whose calls are executed in parallel, $\mathrm{exp}_1[i], \ldots, \mathrm{exp}_n[i]$ where $i \in Ind$ are the actual parameters of the parallel calls of the operation **op**. $f_{\mathbf{Inst}}$ is an auxiliary parameter which is used to define the frame of the parallel composition execution. We explain the usage of $f_{\mathbf{Inst}}$ in **Section 6.3**. The main idea behind the parallel composition is that if there are several operation calls whose effects are disjoint then the order of their execution does not matter and we can execute them simultaneously.

Since the order of the operation execution does not matter the semantics of the parallel composition is equivalent to
$\mathbf{op}(\mathrm{exp}_1[i_1], \ldots, \mathrm{exp}_n[i_1]); \ldots, \mathbf{op}(\mathrm{exp}_1[i_m], \ldots, \mathrm{exp}_n[i_m])$ where
$I = \{i_1, \ldots, i_m\}$. In other words the parallel composition is equivalent to the sequential composition of an unbounded number of operation calls.

The above interpretation of parallel composition is valid only under the assumption that the effects of the operation calls are disjoint. Therefore for each parallel composition we check the proof obligation that for each disjoint $i, j \in I$ $\mathbf{op}(\mathrm{exp}_1[i], \ldots, \mathrm{exp}_n[i])$ and $\mathbf{op}(\mathrm{exp}_1[j], \ldots, \mathrm{exp}_n[j])$ affect disjoint stereotype instances.

The last operation is `skip`. It is an identical transformation of an input slice. It is used when we want to specify that an operation does not change a stereotype slice.

Let us consider an example of composite operations. On Figure 3.13 we define operation `acquireBlocked`. The operation transfers a *source* to the stereotype slice in which *newInstEl* participates. If *source* is **null** then the operation does nothing. Otherwise the basic stereotype operations of `RelationInversion` are used to remove *source* from the old stereotype instance and to add it to the new one. `acquireBlocked` is a partial operation. For instance, it is undefined if *newInstEl* is **null**. The transformation provided in **Section 6.4** can be used to compute the pre-condition of `acquireBlocked`. The pre-condition defines on which values of input variables and stereotype slices the operation `acquireBlocked` is defined.

```
acquireBlocked⟨RelationInversion⟩
    (source: ref, newInstEl: ref)
{
  if (source = null)
  then skip
  else {
    removeRelationInversion(source);
    addRelationInversion(source, newInstEl.sink);
  }
}
```

Figure 3.13: `acquireBlocked` stereotype operation.

3.4 Usage of stereotypes for source code specification

In this section we consider applications of stereotypes for the specification of heap structures and semantic relations between objects.

As an underlying programming language we chose **Java** [55], but the described methodology can be used for any other object oriented language without pointer arithmetic.

We build our specification and verification methodology on the top of a basic verification methodology. We assume that the basic verification methodology provides facilities to specify pre-conditions, post-conditions, method frames, assumptions, assertions, and loop invariants. We also assume that the basic verification methodology provides a formal heap definition and other auxiliary definitions. An example of such a methodology is provided by the **Boogie** [75] tool. **Boogie** is an intermediate verification language based on the weakest pre-condition calculus [43]. **Boogie** is used by many verification tools as an underling verification tool. We also used **Boogie** for our experiments which are described in **Section 5.5**.

A heap can be formalized as a polymorphic map from a pair $\langle o, f\rangle$ to a value of type T, where o is a reference and f is a field of type T. In specifications we denote the value of a field f of an object o in heap h as $h[o.f]$. If the heap is clear from the context then we omit the heap and write just $o.f$. We call $o.f$ a location. A heap can contain ghost fields. A ghost field is used only for specification and can be dropped during compilation of the source code. We define a ghost field by adding the keyword `ghost` to its definition. A predicate `isAllocated`(h, o) checks if the reference o points to an allocated object in heap h. For each heap h `isAllocated`$(h, \textbf{null})$ is false. Function `typeOf`(h, o) returns the dynamic type of the object which is pointed to by the reference o. For each heap h the value of `typeOf`(h, o) is undefined if o points to a non-allocated object or equal to **null**.

Pre- and post- condition definitions have to be provided by a user. They are located in the program source code between a method header and the method body. A pre- and post- condition definition consists of the keywords `requires` and `ensures`, respectively, and an **FOL** formula. A pre-condition can depend on the values of heap locations before the method execution. A post-condition can depend on the values of both heap locations before and after the method execution. Function `old` is used to access a value of a location on a heap before the method execution. For instance, an expression $o.f$ in a post-condition is a value of the field f of the object o after the method execution; an expression `old`$(o.f)$ in a post-condition is a value of the field f of the object o before the method execution. For the method body the semantics of pre- and post- conditions can be described in the following way. A pre-condition is a property which can be assumed before the method execution. A post-condition is a property which has to hold after the method execution if before the method execution all pre-conditions of the method hold. For the method call the semantics of pre- and post- conditions can be described in the following way. A pre-condition is a property which has to hold before the method execution. A post-condition is a property which can be assumed after the method execution if before the method execution all pre-conditions of the method hold.

A special kind of post-condition is the frame specification. The frame specifies which heap locations can be affected by the method invocation. The frame specifications are also located in the program source code between the method header and the method body. A frame specification consists of the `frame` keyword followed by a set of locations. A method execution can change only locations which are mentioned in the frame. A set of locations is a set of pairs of reference and class field name. We use the following notation to describe a set of locations. $o.f$ where o is a reference and f is a field name denotes the singleton set $\{\langle o, f\rangle\}$. $R.f$ where R is a set of references and f is a field name denotes the set $\{o | o \in R \wedge \langle o, f\rangle\}$. $S_1 \cup S_2$ denotes the union of sets S_1 and S_2. $S \times \{f_1, \ldots, f_n\}$ where S is a set of locations and $f_1, \ldots, f_n$ are fields denotes the following set of locations $\bigcup_{i=1}^{n} S.f_i$. By $*$ we denote the set of all fields of an object. For instance, $o.*$ is the set of locations $\{i \in [1..n] | \langle o, f_i \rangle\}$ where the set of fields of object o is $\{f_1, \ldots, f_n\}$.

An assumption is used to assume the validity of an **FOL** formula at a program point. An assumption specification is the keyword `assume` followed by an **FOL** formula φ. The validity of φ is assumed at the program point and can be used to verify the validity of other specifications. For instance we can specify pre-conditions with the help of assumptions. The validity of a pre-condition can be assumed At the beginning of the method body.

An assertion is used to check the validity of an **FOL** formula at a program point. An assertion specification is the keyword `assert` followed by an **FOL** formula φ. The validity of φ is checked at the program point. We can specify post-conditions with the help of assertions. The validity of a post-condition has to be asserted at all exit points of the method.

A loop invariant is a property which has to hold before a loop iteration and has to be preserved by the loop body. A user has to provide loop invariants for each loop to enable the weakest pre-condition calculus [43] based verification. A loop invariant is located in the program source code between the loop header and the loop body. A loop invariant consists of the `Loop Invariant` keyword followed by an **FOL** formula. A special kind of loop invariant is loop frames. A loop frame specification describes which locations are possibly affected by the loop execution. If h is the heap before the loop iteration, h' is the heap after the execution of the loop, and the location $o.f$ does not belong to the loop frame then $h[o, f] = h'[o, f]$. A loop frame is assumed before the loop body execution and has to hold after the loop body execution. A loop frame specification is located in the program source code between a loop header and the loop body. A loop frame specification consists of the `frame` keyword followed by a set of locations.

A program annotated with the above specifications can be verified with the help of the weakest pre-condition calculus [43] (denoted as **WPC**). For a given annotated program **WPC** generates proof obligations. Proof obligations are logical properties which hold if and only if the program satisfies the user provided specification. An automatic theorem prover is used to check the validity of proof obligations. For instance, **Boogie** uses Z3 [39] to verify the generated proof obligations.

We define our stereotype-based technique as an extension of the basic verification technique which is described above. With this intention we introduce several specification primitives which use stereotype slices defined in **Section 3.2** to specify heap properties in a single program point and stereotype operations defined in **Section 3.2.3** to specify heap transformation between different program points. To illustrate the application of the introduced verification concepts we consider the specification and verification of an implementation of the observer pattern with the help of the

RelationInversion stereotype.

```
createObserver⟨RelationInversion, Sequence⟩(o: ref)
{
    createSource(o);
    createSingletonSequence(o);
}

addSeqBegin⟨Sequence⟩(o : ref, o′ : ref)
{
    if (o' ≠ null) addSequenceRelation(o, o');
}

removeSeqElement⟨Sequence⟩(o : ref)
Local variables:
    nextEl = old(o.next);
    prevEl = old(o.prev);
{
    if( nextEl ≠ null) then removeSequenceRelation(o);
    if( prevEl ≠ null) then removeSequenceRelation(o.prev);
    if( prevEl ≠ null ∧ prevEl ≠ null)
        then addSequenceRelation(prevEl, nextEl);
}

function Reg seqElements⟨Sequence⟩(o : ref)
    := if(o = null) then ∅ else o.Elements

function Reg reachable⟨Sequence⟩(o : ref)
    := if(o = null) then ∅ else o.Next*∪ {o}
```

Figure 3.14: Auxiliary stereotype operations and functions.

To specify the implementation of the design pattern we need one more stereotype, Sequence. We use the Sequence stereotype to specify sequences of objects, e.g., various linked lists or a path in a tree. We define the stereotype Sequence in **Section 4**. Here we just list the stereotype items, operations, and predicates which we need for the specification of the observer design pattern implementation.

As any other stereotype Sequence has a stereotype item *Elements* which contains the objects which participate in the same stereotype instance. Stereotype items next and prev refer to the successor and the predecessor of an element of the sequence. Next* of an object o contains the set of the elements which are reachable via the transitive closure of next starting from o. `first` is the first element of the sequence.

The stereotype constructor of `Sequence` is `createSingletonSequence`. It creates a singleton sequence instance. The two basic stereotype operations of the `Sequence` stereotype are `addSequenceRelation` and `removeSequenceRelation`. The first one merges two sequence stereotype instances into one by establishing a `next` relation between the last element of the first stereotype instance and the first element of the second stereotype instance. `removeSequenceRelation` is the inverse operation of `addSequenceRelation`. It splits a stereotype instance in two independent stereotype instances by cutting the `next` relation between two consequent elements of the stereotype instance.

The predicate `isSingletonSequence` checks if a stereotype instance is a singleton sequence.

To abbreviate specifications we introduce several composite stereotype operations and functions. Their definitions are provided in Figure 3.14. `createObserver` combines the constructors of `RelationInversion` and `Sequence` stereotypes. Below we use `createObserver` to specify the constructor of the observer.

We use the operation `addSeqBegin` to add an object to the beginning of a sequence stereotype instance. The first input parameter represents the added object and the second one the first element of the sequence to which the object is attached. It is possible that the sequence is empty. In this case the second parameter is null and the operation does not change the stereotype slice. Otherwise the operation executes `addSequenceRelation`. `removeSeqElement` removes an element from the sequence. The only input parameter refers to the removed element. If the removed element is not the last one in the sequence the operation removes the relation between the removed element and the next one. In a similar way if the removed element is not the first one in the sequence the operation removes the relation between the removed element and the previous one. If the removed element is both not the first one and not the last one the operation merges the remaining sequence tails into a one sequence.

The function `seqElements` returns the elements of the stereotype instance to which the input element belongs. If the input element is null then the function returns the empty set. The function `reachable` returns the set of elements which are reachable via the transitive and reflexive closure of `next` starting from the input parameter o. If the input parameter is null then the function returns the empty set.

Let us now consider the implementation of the observer pattern. The implementation consists of classes **Observer** and **Subject**. At first we describe the implementation of **Observer** and **Subject**. The implementation description refers to the figures which contain both the implementation and the specification of the design pattern. Some of the specification primitives which we use to specify the design patten are explained later.

```
enum ObserverState = {st_Detached , st_OnConstruction , st_Attached}

class Observer participate ⟨RelationInversion : ri, Sequence⟩{
    int value;
    Observer nextObs, prevObs;
    ghost ObserverState st;

    Glue Invariant st = st_Detached ⇒isSingletonSequence(this);
    Glue Invariant st = st_Detached ⇒isSingletonSource(this);
    Glue Invariant st ≠ st_Detached ⇒ nextObs = next;
    Glue Invariant st ≠ st_Detached ⇒ prevObs = prev;

    Observer()
    stereotype createObserver(this);
    ensures st = st_Detached;
    {
        createObserver(this);
        st = st_Detached;
    }

    void update(Subject sub)
    requires sub ≠ null;
    requires this.sink = sub;
    requires invAllAttached(sub);
    ensures invObserver(this);
    frame this.value;
    {
        value = sub.value;
    }
}
```

Figure 3.15: Definition and specification of the class **Observer**.

```
void add(Observer o)
requires st = st_{Detached};
requires o ≠ null ⇒ o.st = st_{Attached};
ensures st = st_{OnConstruction};
frame this.nextObs ∪ o.prevObs ∪ this.st;
stereotype addSeqBegin(this, o);
{
    if(o != null){
        nextObs = o;
        o.prevObs = this;
        addSequenceRelation(this, o);
    }
    st = st_{OnConstruction};
}

void remove()
requires st = st_{OnConstruction};
requires nextObs ≠ null ⇒ nextObs.st = st_{Attached};
requires prevObs ≠ null ⇒ prevObs.st = st_{Attached};
ensures st = st_{Detached};
frame nextObs.prevObs ∪ prevObs.nextObs ∪ this.st;
stereotype removeSeqElement(this);
{
    if(nextObs != null)
        nextObs.prevObs = prevObs;
    if(prevObs != null)
        prevObs.nextObs = nextObs;
    removeSeqElement(this);
    st = st_{Detached};
}
```

Figure 3.16: Definition and specification of *add* and *remove* methods of the class **Observer**.

```
enum SubjectState = {st_Head , st_NotHead}

class Subject participate ⟨RelationInversion : ri⟩{
    int value;
    Observer head;
    ghost SubjectState st;

    Glue Invariant st = st_Head ∧ head ≠ null ⇒
        head = head.first;
    Glue Invariant seqElements(head) ⊆ this.Source;
    Glue Invariant ∀ o ∈ this.Source \ seqElements(head) :
        o.st ≠ st_Attached;

    Subject()
    ensures invSubject(this);
    stereotype createSink(this);
    {
        createSink(this);
        st = st_Head;
    }

    void update(int newValue)
    requires invSubject(this);
    ensures invSubject(this);
    frame this.Source.value ∪ this.value;
    {
        this.value = newValue;
        Observer it = head;
        while(it ≠ null)
        Loop Invariant invSubjectSt(this,reachable(it));
        frame (this.Source\ reachable(it)).value;
        {
            it.update(newVal);
            it = it.nextObs;
        }
    }
}
```

Figure 3.17: Definition and specification of the class **Subject**.

```
void attach(Observer observer)
requires observer.st = st_Detached;
requires invSubject(this);
ensures invSubject(this);
frame observer.nextObs ∪ head.prevObs ∪ observer.st
    ∪ this.head;
stereotype addRelationInversion(this, observer);
stereotype addSeqBegin(this, o);
{
    addRelationInversion(observer, this);
    st = st_NotHead;
    observer.add(head);
    head = observer;
    st = st_Head;
    observer.st = st_Attached;
    observer.update(this);
}

void detach(Observer observer)
requires invSubject(this);
ensures invSubject(this);
ensures observer.st = st_Detached;
frame observer.nextObs.prevObs ∪ this.head ∪ this.st
    ∪ observer.prevObs.nextObs;
stereotype removeSeqElement(observer);
stereotype removeRelationInversion(observer);
{
    observer.st = st_OnConstruction;
    st = st_NotHead;
    observer.remove();
    removeRelationInversion(observer);
    if (head = observer)
        head = observer.nextObs;
    st = st_Head;
}
```

Figure 3.18: Definition and specification of the methods *attach* and *detach* of the class **Subject**.

The implementation of class **Observer** is shown on Figure 3.15 and Figure 3.16. The class header also defines in which stereotype slices the class **Observer** participates. We consider the participation of a class in a stereotype slice in **Subsection 3.4.1**.

The field *value* contains a copy of the corresponding field of the **Subject**. The value of *value* is the data which has to be synchronized for the subject and observers. Below we formalize the invariant which states that the value of the fields *value* has to be the same for the subject and all attached observers.

Fields *nextObs* and *prevObs* are used to organize observers in a doubly-linked list. The list is used by the subject to track all observers which are attached to the subject. When an observer is attached to the subject the observer added to the list and removed when it is detached.

Ghost field `st` is a ghost field which is used to define the glue invariant. The definition of the glue invariant follows immediately after the definition of `st`. Glue invariants are used to bind together stereotype slices and the program heap. We consider glue invariants and states in **Subsection 3.4.3**. The value of the field `st` changes during the program execution. It can be assigned like a normal object field. These updates are related to glue invariants and we also consider them in **Subsection 3.4.3**.

The constructor of the **Observer** creates an observer which is not attached to a subject. The body of the constructor contains only specification primitives but no source code. The first line of the constructor contains a call of the stereotype operation `createObserver`. We use stereotype operations in the source code to specify the update of values of stereotype items. Since such updates do not affect values of non-ghost fields. We consider the usage of stereotype operations for ghost updates in **Subsection 3.4.2**.

Another stereotype construct which is used in the constructor of **Observer** is a stereotype method specification. This specification begins with the keyword `stereotype`. It uses stereotype operations to specify the method behavior. We consider stereotype method specifications in **Subsection 3.4.4**.

The method *update* is executed by the subject to which the observer is attached. The method notifies the observer regarding an update of the subject. In response to the notification, the method synchronizes the value of the field *value* with the value of the subject. The method *add* adds **this** to the linked list whose first element is the input parameter *o*. If *o* is null then the operation does nothing. The method *remove* removes **this** from the linked list to which **this** currently belongs.

The implementation of class **Subject** is shown on Figure 3.17 and Figure 3.18. The field *value* contains the data which has to be synchronized between the subject and observers. The field *head* contains a reference to the list of observers which are attached to the subject. If there are no observers attached to the subject then *head* is null. The last field `st` is a ghost field which is used as glue state.

The constructor creates a subject to which no observers attached. The body of the constructor contains only specification primitives but not real source code.

The method *update* updates the value of *value* and notifies all attached observers. At first the method sets the new value of the field *value*. Then the method notifies all the attached observers. To notify all the attached observers the method declares a local variable *it* of type **Observer** and sets it to the head of the list of attached observers. Then the loop iterates over all attached observers and calls *update*. The loop terminates when *it* is equal to null.

The method *attach* attaches the observer referred to by the input parameter. First the method calls *add* of the *observer* to add the attached observer to the list of attached observers. Since the new head of the list is *observer* the method sets the field *head* to *observer*. At the end the method calls *update* of the connected observer.

The method *detach* is an inversion of the method *attach*. It detaches the observer referred to by the input parameter *observer* from the subject. First the method calls *remove* of *observer* to remove the attached observer from the list of the attached observers. If *observer* is equal to *head* then the execution of *observer.remove()* changes the first element of the list of the attached observers. In this case the method sets the field *head* to *observer.nextObs*.

`Bool invObserver`(o: **Observer**)
$\texttt{invObserver}(o) \Leftrightarrow o.value = o.sink.value$

`Bool invSubjectSt`(s: **Subject**, R: *Reg*)
$\texttt{invSubjectSt}(s, R) \Leftrightarrow \forall o \in s.Source \setminus R : \texttt{invObserver}(o)$

`Bool invAllAttached`(s: **Subject**)
$\texttt{invAllAttached}(s) \Leftrightarrow \forall o \in s.Source : o.\texttt{st} = \texttt{st}_{Attached}$

`Bool invSubject`(s: **Subject**)
$\texttt{invSubject}(s) \Leftrightarrow \texttt{invSubjectSt}(s, \varnothing) \wedge \texttt{invAllAttached}(s) \wedge s.\texttt{st} = \texttt{st}_{Head}$

Figure 3.19: Behavioral invariants of **Subject** and **Observer** classes

Let us now consider the properties of the observer design pattern implementation which we would like to verify. As we have mentioned above we would like to verify that the value of of the field *value* has to be the same for a subject and all observers which are attached to the subject. We formalize this property using predicates whose definitions are shown in Figure 3.19. $\texttt{invObserver}(o)$ holds if and only if the value of the field *value* of o is equal to the value of the subject to which o is attached. In other words the observer o is synchronized with its subject. `invSubjectSt`(s, R) holds if and only if all observers which are attached to the subject s and do not belong to the set R are synchronized with the subject. The predicate `invAllAttached` has a technical nature and we consider it later. `invSubject` summarizes the above properties. Among others it contains `invSubjectSt`$(s, \varnothing)$ which states that all observers which are attached to the subject s are synchronized with the subject. We call the above properties behavioral invariants.

Essentially we would like to guarantee preservation of the behavioral invariants for all instances of the design pattern. For instance, if there are three instances of the observer pattern and one of them attaches an observer we would like to be sure that the behavioral invariant is preserved for all three instances. Verification of the behavioral invariant preservation relies on topological properties, e.g., observers form an acyclic doubly-linked lists, the field *head* of an observer contains the head of a list, different subjects refer to different lists. We consider the specification and verification of behavioral invariants in **Section 3.4.5**.

The rest of the section is organized as follows.

In **Subsection 3.4.1** we describe participation of a class in stereotype slices. In **Subsection 3.4.2** we describe how stereotype operations can be used to specify ghost updates of stereotype items. **Subsection 3.4.3** describes how stereotype slices and program heap can be bound together with the help of glue invariants. **Subsection 3.4.4** describes how stereotype operations can be used to specify class methods and loop invariants. This section is concluded by **Subsection 3.4.5** which specifies how on top of topological heap properties behavioral invariants can be specified and verified.

3.4.1 Participation of a class in a stereotype slice

We use the notation *Cl* `participate` $\langle St_1 : StName_1, \ldots, St_n : StName_n\rangle$ to specify that a class *Cl* participates in stereotype slices $StName_1, \ldots, StName_n$ of types $St_1, \ldots, St_n$. It is possible that several classes participate in the same stereotype slice. Such shared stereotype slices identify a semantic relation between objects. For instance, **Subject** and **Observer** share the same stereotype instance `RelationInversion` : ri (see Figure 3.15 and Figure 3.17). Therefore from the information which is provided by class headers (both of them use the same stereotype name ri) we can conclude that there is a semantic relation between **Subject** and **Observer** which is described by the stereotype `RelationInversion`. In this way we can explicitly specify that there is a semantic relation between different classes and specify the nature of this relation.

Quite often a class does not share a stereotype slice of type *St* with other classes and the class participates in only one stereotype slice of type

St. In this case we allow one to omit the stereotype slice name. We call this stereotype slice unnamed. For instance class **Observer** (see Figure 3.15) participates in an unnamed stereotype slice of type `Sequence`. This slice is used to organize objects of type **Observer** into doubly-linked lists. The objects of other types do not participate in this slice. For each unnamed stereotype slice of type *St* which participates in class *Cl* we implicitly add the name $Cl :: St$. Since *Cl* participates in the only one unnamed stereotype slice of type *St* the generated name is unique. By doing so we reduce all unnamed stereotype slices to named stereotype slices.

A class can participate in several stereotype slices of the same type. For instance, in **Section 5.1** we provide a specification of an implementation of a list with iterator. The class **List** (see Figure 5.1) participates in two stereotype slices of type `RelationInversion`. One is used to specify the semantic relation between the list and its iterator, another one is used to specify the ownership-like relation between the list and the list nodes.

We introduce predicate $\texttt{partIn}(Cl, StName)$ which holds if and only if the class *Cl* participates in the stereotype slice *StName*. For a given program the predicate can be constructed from the class headers.

From the class header we can extract information about which stereotype slices have to be declared for a given program. A stereotype slice $St : StName$ of type *St* has to be declared for a given program if and only if the program contains at least one class *Cl* such that $\texttt{partIn}(Cl, StName)$ holds. We assume that all classes which participate in a stereotype slice declare it with the same type *St*. The validity of the last property can be checked by examination of class headers.

The values of stereotype items are used in the program specification. If *exp* is an expression of type *Cl*, *StName* is a stereotype slice of type *St*, and *StItem* is an item of the stereotype *St* then we denote access to the *StItem* of the stereotype slice *StName* of the object which is referred by the expression *exp* as *exp.StName.StItem*. If there is only one stereotype slice of type *St* such that *St* has stereotype item *StItem* then the stereotype slice can be omitted form the access expression and the access expression can be described as *exp.StItem*. For instance, since class **Subject** participates in only one stereotype slice of type `RelationInversion` which name is *ri* we can abbreviate *s.ri.Source* as *s.Source* where *s* is a variable of type **Subject**.

Recall that according to **Definition 4** an object *o* participates in a stereotype slice *StName* if and only if $o \in \mathbf{Dom}(StSlice)$. On the other hand in this section we have defined participation of a class *Cl* in a stereotype slice *StSlice*. The consistency between participation of an object *o* of type *Cl* in a stereotype slice *StName* and participation of class *Cl* in the stereotype slice *StName* is formalized by the following definition.

Definition 18 (System invariant about the participation in a stereotype slice.)**.** *We say that the system invariant about the participation in a stereo-*

type slice holds for a stereotype slice StName *and a heap* h *(denoted by* $\boldsymbol{SysInvPart}[\mathrm{StName}, \mathrm{h}]$*) if*
$\forall o : o \in \boldsymbol{Dom}(\mathrm{StName}) \Leftrightarrow$
$\mathtt{isAllocated}(h, o) \wedge \mathtt{partIn}(\mathtt{typeOf}(h, o), \mathrm{StName})$
An object o participates in a stereotype slice StName *if and only if o is allocated and the class of o participates in the stereotype slice* StName*.*

In the next subsection we prove that for each stereotype slice the above invariant holds in each program point.

3.4.2 Ghost updates

In this subsection we consider how stereotype operations can be used to specify ghost updates of stereotype items during the program execution. In **Section 6.4** we provide a transformation of stereotype operations to a pair of pre- and post- conditions. In this section we rely on this transformation and treat a stereotype operation as a pair of pre and post conditions. In **Section 6.4** it is also proven that for each affected stereotype slice *StName* each stereotype operation preserves the stereotype system invariant **SysInv**[*StSlice*]. We use the above property to prove that for each stereotype slice the stereotype system invariant holds at each program point.

Each stereotype operation header contains a list of input parameters and a list of affected stereotype slices. Both the input parameters and the affected slices have to be instantiated when the stereotype operation is executed in the source code. For instance, the header of the stereotype operation `addRelationInversion` is
`addRelationInversion`$\langle$`RelationInversion`$\rangle$(*source* : *ref*, *sink* : *ref*).
Therefore the values of the input parameters *source* and *sink* and the value of the affected unnamed stereotype slice of type `RelationInversion` have to be instantiated. A valid call of `RelationInversion` from source code could look like `addRelationInversion`$\langle ri \rangle (v, u)$ where v and u are variables of a reference type and ri is a name of a stereotype slice of type `RelationInversion` which is defined in the verified program. For each call we check that the provided stereotype slices have proper types. We also check that if an operation call affects to stereotype slices $StName_1$ and $StName_2$ of the same type then $StName_1 \neq StName_2$. Since there is no aliasing between stereotype slices the last check is sufficient to guarantee that the operation call does not affect the same stereotype slice twice.

In most cases a method calls stereotype operations to update a stereotype slice in which the enclosing class participates. Therefore if a stereotype operation affects the stereotype slice *StName* of type *St* and the enclosing class participates in only one stereotype slice of type *St* then the name*StName* can be omitted from the **op** call. For instance, method *attach* of class **Subject** (see Figure 3.18) makes the following stereotype operation call `addRelationInversion`$(observer, \mathbf{this})$. Since `addRelationInversion` affects a stereotype slice of type `RelationInversion` and class **Subject** participates in only one stereotype slice of type `RelationInversion` ri, `addRelationInversion`$(observer, \mathbf{this})$ is an abbreviation for `addRelationInversion`$\langle ri \rangle(observer, \mathbf{this})$.

In **Section 6.4** we will describe the procedure which transform a given stereotype operation into a pair of pre- and post- conditions. We use these pre- and post- conditions to define the semantics of a stereotype operation call from the source code.

Definition 19 (Semantics of a stereotype operation call)**.** *If*

- *the header of a stereotype operation* $\boldsymbol{op}$ *is* $\langle \mathrm{St}_1 : \mathrm{StName}_1^{\boldsymbol{op}}, \ldots, \mathrm{St}_n : \mathrm{StName}_n^{\boldsymbol{op}} \rangle \boldsymbol{op}(v_1 : T_1, \ldots, v_m : T_m)$
- *the pre-condition of* $\boldsymbol{op}$ *is the* ***FOL*** *formula* $\mathbf{pre}$ *which depends on*
  - *formal parameters* $v_1, \ldots, v_m$
  - *the states of the affected stereotype slices* $\mathrm{StName}_1^{\boldsymbol{op}}, \ldots, \mathrm{StName}_n^{\boldsymbol{op}}$ *before the operation execution*
- *the post-condition of the* $\boldsymbol{op}$ *is the* ***FOL*** *formula* $\mathbf{post}_{\boldsymbol{op}}$ *which depends on*
  - *formal parameters* $v_1, \ldots, v_m$
  - *the states of the affected stereotype slices* $\mathrm{StName}_1^{\boldsymbol{op}}, \ldots, \mathrm{StName}_n^{\boldsymbol{op}}$ *before the operation execution*
  - *the states of the affected stereotype slices* $\mathrm{StName}_1'^{\boldsymbol{op}}, \ldots, \mathrm{StName}_n'^{\boldsymbol{op}}$ *after the operation execution*

then the semantics of the operation call $\langle \mathrm{StName}_1, \ldots, \mathrm{StName}_n \rangle \boldsymbol{op}(exp_1, \ldots, exp_m)$ *where*

- $\mathrm{StName}_1, \ldots, \mathrm{StName}_n$ *are the stereotype slices affected by the operation call*
- $exp_1, \ldots, exp_m$ *are the actual parameters of the operation call*

can be defined as:

1. *To each stereotype slice* StName_i *we assigned a new value denoted as* StName_i' *where* $i \in [1..n]$*.*
2. *Before the operation execution the assertion* `assert` $\mathbf{pre}_{\boldsymbol{op}}'$ *is checked, where* $\mathbf{pre}_{\boldsymbol{op}}'$ *is constructed from* $\mathbf{pre}_{\boldsymbol{op}}$ *by replacing all occurrences of* v_i *by* exp_i *and* $\mathrm{StName}_j^{\boldsymbol{op}}$ *by* StName_j *where* $i \in [1..m]$ *and* $j \in [1..n]$*.*

3. *After the operation execution the assumption* `assume` $\boldsymbol{post'_{op}}$ *is assumed, where* $\boldsymbol{post'_{op}}$ *is constructed from* $\boldsymbol{post_{op}}$ *by replacing all occurrences of* v_i *by* exp_i*,* StName_j^{op} *by* StName_j*, and* $\mathrm{StName}_j'^{op}$ *by* StName'_j *where* $i \in [1..m]$ *and* $j \in [1..n]$*.*

As an example of the stereotype operation call let us consider the call `addRelationInversion`$(observer, \mathbf{this})$ form the method *attach* of class **Subject** (see Figure 3.18). As we have mentioned above `addRelationInversion`$(observer, \mathbf{this})$ is an abbreviation of `addRelationInversion`$\langle ri\rangle(observer, \mathbf{this})$. The operation execution is equivalent to:

- Assignment to the stereotype slice ri of a new value which is denoted as ri'
- Assertion of the operation pre-conditions
  - $\mathbf{this}.ri.sink = \mathbf{this}$ generated from the first operation pre-condition.
  - $observer.ri.sink = \mathbf{null}$ generated from the second operation pre-condition
  - $observer.ri.\mathtt{instID} \neq \mathbf{this}.ri.\mathtt{instID}$ checks that input instances are disjoint
- Assumption of the post condition:
  - $observer.ri'.sink = \mathbf{this}$ generated from the first transformation rule.
  - $\forall o : o \neq observer \Rightarrow o.ri'.sink = o.ri.sink$ the frame property generated from the first transformation rule.
  - $\forall o : o \in observer.ri.Elements \cup \mathbf{this}.ri.Elements \Rightarrow$ $o.ri'.Source = \mathbf{this}.ri.Source \cup \{observer\}$ generated from the second transformation rule.
  - $\forall o : o \notin observer.ri.Elements \cup \mathbf{this}.ri.Elements \Rightarrow$ $o.ri'.Source = o.ri.Source$ the frame property generated from the second transformation rule.

If the post-condition is inconsistent then the soundness of the verification can be violated. For instance, if a post-condition of a stereotype operation is $false$ and **op** is called from the body of a method m then the method m would be verified for any post-condition including post-condition $false$. Therefore in **Section 6.4** we also prove for each stereotype operation that if the operation pre-condition holds then the operation post-condition is feasible. In this way we guarantee that a stereotype operation execution does not violate soundless of the method verification.

Stereotype constructors play a special role and therefore their execution differs from the execution of normal stereotype operations. Below we define the syntactic constraints imposed on the verified program. One of the constraint states that the stereotype constructors can be executed only in the beginning of the class constructors. A stereotype constructor is executed immediately after a new object allocation but before the execution of the constructor of the object. In other words all stereotype constructors calls are completed before the class constructor execution.

In case of the inheritance, all stereotype constructors are executed before the the execution of the base constructor. As an example let us consider the following situation: a class A participates in a stereotype slice $StName_A$ and a subclass B of the class A participates in a stereotype slice $StName_B$. The constructor of the class A executes a stereotype constructor $\mathbf{op}_A$ and the the constructor of class B executes a stereotype constructor $\mathbf{op}_B$. If a new object of type A is allocated then:

- the stereotype constructor $\mathbf{op}_A$ is executed
- the constructor of class A is executed

If a new object of type B is allocated then:

- the stereotype constructor $\mathbf{op}_A$ is executed
- the stereotype constructor $\mathbf{op}_B$ is executed
- the constructor of class A is executed
- the constructor of class B is executed

Thus we can be sure that the stereotype system invariant is enforced for a fresh allocated object before the beginning of the class constructor execution.

In **Section 6.4** we will proven that for each affected stereotype slice *StName* each stereotype operation preserves the stereotype system invariant **SysInv**[*StSlice*]. We are using this property of stereotype operations to guarantee that for each stereotype slice which is declared by the program the stereotype system invariant holds at every program point. On the other hand a direct update of a stereotype slice can violate the stereotype system invariant. Therefore we forbid direct updates of stereotype items.

To guarantee that the stereotype system invariant holds at every program point we have not only to preserve the invariant but also establish it for freshly allocated objects. As we have mentioned in **Subsection 3.3.1** we use stereotype constructors to establish the stereotype system invariant for the freshly allocated objects. Therefore we require that the body of each constructor of a class *Cl* begins with the stereotype constructors for each stereotype slice in which class *Cl* participates. In this way we also can guarantee the preservation of **SysInvPart**$[StName, h]$ which we introduced in **Subsection 3.4.1**.

Let us combine the syntactic constraints which we impose on a verified program.

Definition 20 (Syntactic constraints imposed on verified programs)**.**

- *Direct updates of stereotype items are forbidden. Values of stereotype items are updated only by stereotype operations.*
- *The body of each constructor of a class* Cl *which participates in stereotype slices* $\mathrm{StName}_1, \ldots, \mathrm{StName}_n$ *of types* $St_1, \ldots, \mathrm{St}_n$ *has to begin with* $\boldsymbol{op}_1(\boldsymbol{this}), \ldots, \boldsymbol{op}_n(\boldsymbol{this})$ *where* $\boldsymbol{op}_1, \ldots, \boldsymbol{op}_n$ *are constructors of stereotypes* $St_1, \ldots, \mathrm{St}_n$*.*
- *A stereotype constructor can be executed only at the beginning of a class constructor.*

The above constraints can be verified by a simple syntactic procedure. Therefore we assume that the above constraints hold for all verified programs.

To prove the preservation of the stereotype system invariant we need a proper description of the initial program state. We describe the initial program state in the following definition:

Definition 21 (Initial program state)**.** *In the initial program state the following holds:*

- $\forall o : \neg \texttt{isAllocated}(\mathrm{h}, o)$ *where* h *is the initial program heap. That is, at the beginning of the program execution there are no allocated objects.*
- *For each stereotype slice* StSlice *defined by the program at the beginning of the program execution the following holds:* $\boldsymbol{Dom}(\mathrm{StName}) = \varnothing$*. That is, at the beginning of the program execution no object participates in any of the program-defined stereotype slices.*

Let us now prove theorem about the preservation of the stereotype system invariant.

Theorem 1 (Preservation of the stereotype system invariant)**.** *At each program point of the program which satisfies the syntactic constraints from* ***Definition 20*** *for each stereotype slice which is declared by the program holds* $\boldsymbol{SysInv}[\mathrm{StName}] \wedge \boldsymbol{SysInvPart}[\mathrm{StName}, \mathrm{h}]$ *where* h *is the program heap at the current program point.*

Proof. The proof is done by induction over the program execution.

Induction base. According to **Definition 21** at the beginning of the program execution none of the objects is allocated and none of the objects participates in any of stereotype slice. Therefore for any program-defined stereotype slice $\mathbf{SysInvPart}[\mathit{StName}, h]$ trivially holds where h is the program heap at the beginning of the program execution.

According to **Definition 21** at the beginning of the program execution $\mathbf{Dom}(\mathit{StName}) = \varnothing$ holds for any program-defined stereotype slice StName. Since all the properties $\mathbf{SysInv}[\mathit{StName}]$ include universal quantification over $\mathbf{Dom}(\mathit{StName})$, $\mathbf{SysInv}[\mathit{StName}]$ trivially holds.

Induction step. Let us first prove the preservation of $\mathbf{SysInvPart}[\mathit{StName}, h]$. According to **Definition 18** $\mathbf{SysInvPart}[\mathit{StName}, h]$ holds if and only if $\forall o : o \in \mathbf{Dom}(\mathit{StName}) \Leftrightarrow \mathtt{isAllocated}(h, o) \wedge \mathtt{partIn}(\mathtt{typeOf}(h, o), \mathit{StName})$. `partIn` does not depend on the program state. Therefore the validity of **SysInvPart** can be affected only by modification of **Dom**, `isAllocated`, and `typeOf`. **Dom** is modified only by stereotype constructor and `isAllocated` and `typeOf` are modified only by an object allocation. According to the syntactic constraints from **Definition 20** an allocation of a new object o of type Cl is always coupled together with the execution of the stereotype constructor $\mathbf{op}(o)$ for each stereotype slice StName in which class Cl participates. We denote the state of the heap and the sate of the stereotype slice before the object allocation and the stereotype constructor execution as h and StName, respectively. We denote the state of the heap and the sate of the stereotype slice after the object allocation and the stereotype constructor execution as h' and StName', respectively. The effects of the allocation of object o and the execution of stereotype constructor **op** which are relevant to $\mathbf{SysInvPart}[\mathit{StName}, h]$ are:

- $\forall o' : o' \neq o \Rightarrow (\mathtt{isAllocated}(h, o') \Leftrightarrow \mathtt{isAllocated}(h', o'))$, $\mathtt{isAllocated}(h, o)$ is false, and $\mathtt{isAllocated}(h', o)$ is true. These relations are established by the allocation of object o.

- $\forall o' : o' \neq o \Rightarrow (\mathtt{typeOf}(h, o') = \mathtt{typeOf}(h', o'))$, $\mathtt{typeOf}(h, o)$ is undefined, and $\mathtt{typeOf}(h', o) = \mathit{Cl}$. These relations are established by the allocation of object o.

- $\mathbf{Dom}(\mathit{StName}') = \mathbf{Dom}(\mathit{StName}) \cup \{o\}$. This relation is established by the execution of stereotype constructor **op**.

- for each class Cl $\mathtt{partIn}(\mathit{Cl}, \mathit{StInst}') = \mathtt{partIn}(\mathit{Cl}, \mathit{StInst})$.

Let us now prove that $\mathbf{SysInvPart}[\mathit{StName}, h]$ implies $\mathbf{SysInvPart}[\mathit{StName}', h']$. For each object o'
$o' \in \mathbf{Dom}(\mathit{StName}') \Leftrightarrow o' \in \mathbf{Dom}(\mathit{StName}) \cup \{o\} \Leftrightarrow$
$(o' \neq o \wedge \mathtt{isAllocated}(h, o') \wedge \mathtt{partIn}(\mathtt{typeOf}(h, o'), \mathit{StName})) \vee o' = o \Leftrightarrow$
$(o' \neq o \wedge \mathtt{isAllocated}(h', o') \wedge \mathtt{partIn}(\mathtt{typeOf}(h', o'), \mathit{StName}'))\vee$
$(o' = o \wedge \mathtt{isAllocated}(h', o') \wedge \mathtt{partIn}(\mathtt{typeOf}(h', o'), \mathit{StName}')) \Leftrightarrow$
$\mathtt{isAllocated}(h', o') \wedge \mathtt{partIn}(\mathtt{typeOf}(h', o'), \mathit{StName}')$.

If we combine the above then we get $\forall o' \in \mathbf{Dom}(\mathit{StName}') \Leftrightarrow$ $\texttt{isAllocated}(h', o') \wedge \texttt{partIn}(\texttt{typeOf}(h', o'), \mathit{StName}')$ which implies $\mathbf{SysInvPart}[\mathit{StName}', h']$.

Let us now prove the preservation of $\mathbf{SysInv}[\mathit{StName}]$. According to the syntactic constraints from **Definition 20** the values of stereotype items are updated only by stereotype operations. On the other hand in **Section 6.4** in **Theorem 30** it is proven that for each affected stereotype slice *StName* each stereotype operation preserves the stereotype system invariant $\mathbf{SysInv}[\mathit{StSlice}]$. Therefore we can conclude that $\mathbf{SysInv}[\mathit{StSlice}]$ is preserved by the program execution. $\square$

Let us now consider how stereotype operations are used to specify the implementation of the observer design pattern which is considered above.

On Figure 3.15 we can see the definitions of the constructor and *update* method of the class **Subject**. The constructor uses stereotype constructors `createSource` and `createSingletonSequence` to initiate the stereotype items of the stereotype slice *ri* of type `RelationInversion` and of the unnamed stereotype slice of type `Sequence`. Both constructors are packed into one stereotype operation `createObserver`. `createSource` creates a singleton stereotype instance in *ri* where a freshly allocated object plays the role of a source. `createSingletonSequence` creates a singleton sequence stereotype instance which is used to specify that the freshly allocated object forms a singleton linked list. The *update* method does not use any stereotype operation calls because it does not change the heap topology.

On Figure 3.16 we can see the definitions of the methods *add* and *remove* of class **Observer**. The method *add* uses `addSequenceRelation` to preserve the consistency between the doubly-linked list formed by the observers and the stereotype slice which is used to specify properties of this list. An execution of `addSequenceRelation` updates the `Sequence` stereotype slice in a way that it reflects the addition of the new element to the beginning of the doubly-linked list. In the similar way method *remove* uses `removeSeqElement` to reflect in the stereotype slice `Sequence` the removal of an element from the doubly-linked list. The execution of `removeSeqElement` moves **this** into a separate stereotype instance. Please note that even though `removeSeqElement` moves **this** into the separate stereotype instance, fields *nextObs* and *prevObs* of **this** still refer to the list elements to which **this** used to belong before the operation execution. We discuss in more detail the relation between the heap fields and stereotype items in **Subsection 3.4.3**.

On Figure 3.17 we can see the definitions of the constructor and method *update* of the class **Observer**. The constructor uses stereotype constructor `createSink` to initiate the serotype items of the stereotype slice *ri*. `createSink` creates a singleton stereotype instance in *ri* where a freshly allocated object plays the role of a sink. The *update* method does not use any stereotype operation calls because it does not change the heap topology.

On Figure 3.18 we can see the definitions of the methods *attach* and *detach* of the class **Subject**. The method *attach* uses `addRelationInversion` to reflect the attachment of the observer *observer* to the subject. In a similar way the method *detach* uses `removeRelationInversion` to reflect the detachment of the observer *observer* from the subject.

You can see that stereotype operation calls are strongly related with field updates. Therefore most of the stereotype operation calls can be inferred from the field updates. In this thesis we describe only the most essential machinery used in stereotype-based verification and do not consider specification inference. Nevertheless we believe that the inference of stereotype operation calls is an interesting direction for the future work.

3.4.3 Glue invariants

Up to now we have considered stereotype specifications independently from the program heap, without connection between the items of a stereotype slice and the program heap fields. To couple together the stereotype items and the heap fields we introduce *glue invariants*. A *glue invariant* is a universally quantified **FOL** formula which depends on the variable **this** and can mention stereotype items of any stereotype slice and fields of any class.

Glue invariants have standard visible state semantics [102]: we assert that all glue invariants hold for all objects at the end of a method body and before a method call; we assume the invariant holds at the beginning of a method body and after a method call. Constructors are treated in a special way. At the beginning of a constructor we assume that all glue invariants hold for all objects excluding the freshly allocated one. In [102] it is proven that the above assumptions are valid. The proof is done by induction over the program execution. We also add the glue invariants of all objects to all loop invariants. We choose the visible state semantics to avoid specification overhead. Otherwise we would have to specify explicitly where we can assume a glue invariant and where we have to assert it.

Nevertheless there are several typical situations when the visible state semantics is not flexible enough. The first typical situation is when a heap structure is rebuilt. For instance if a forward link of a double linked list is updated but the respective backward link still has an old value and we would like to make a method call between these two updates. Another typical situation which displays a flexibility problem is a topology change. For instance in the **PIP Subsection 5.4** example we will see that a node can dynamically switch from the participation in a tree to the participation in a cyclic list.

To address these flexibility issues we introduce *glue invariant states*. A glue invariant state is a ghost field of an enumeration type whose elements correspond to different topological states of a class instance. For instance, in the list example we could have a glue invariant state with two states.

When an object is in the first glue invariant state a backward link is broken. When an object is in the second glue invariant state both the forward and the backward links are consistent with the respective sequence stereotype slice. In the **PIP Subsection 5.4** example we also have two states: one corresponds to the situation when a node participates in a tree, and the other one when a node participates in a cyclic list. We use a glue invariant and stereotype items to establish relations between glue invariant states of different objects.

Here we intentionally do not put any restrictions on the classes and stereotype slices which a glue invariant can mention. By this we try to avoid introducing of extra machinery and keep our approach relatively simple.

Let us now consider how the glue invariants and glue states are used to specify the implementation of the observer design pattern.

On Figure 3.15 we can see the definition of the **Observer** glue states and glue invariants. The glue state of **Observer** is represented by the field `st` of type **ObserverState**. The values of type **ObserverState** represent possible topological states of the observer:

- $\texttt{st}_{Detached}$ - the observer is detached from a subject.
- $\texttt{st}_{Attached}$ - the observer is attached to a subject.
- $\texttt{st}_{OnConstruction}$ - the observer in an intermediate state between attached and detached.

The **Observer** has four glue invariants. The first two glue invariants state that if an observer is in the state $\texttt{st}_{Detached}$ then the observer forms a singleton stereotype instance in both the `Sequence` and the *ri* stereotype slice. The last two glue invariants state that if the observer is in the state $\texttt{st}_{OnConstruction}$ or $\texttt{st}_{Attached}$ then fields *nextObs* and *prevObs* are equal to the respective stereotype items of the `Sequence` stereotype slice. Since stereotype items `next` and `prev` form an acyclic sequence the same properties are inherited by *nextObs* and *prevObs*. In this way we can be sure that *nextObs* and *prevObs* form a doubly-linked list.

At the beginning of the constructor of **Observer** the validity of the invariant is assumed for all the objects excluding **this**. The constructor sets the glue state of the freshly allocated object to $\texttt{st}_{Detached}$. At the end of the constructor the validity of all glue invariants is checked for all of the allocated objects. The glue invariants of all allocated objects excluding **this** are preserved. Since the state of **this** is $\texttt{st}_{Detached}$ we have to prove that **this** forms the singleton instances in the `Sequence` and the *ri* stereotype slice. This immediately follows from the post-condition of the call of stereotype operation `createObserver`. The post-condition of the constructor states that the state of the freshly allocated object is $\texttt{st}_{Detached}$.

The method *updated* does not change the heap topology and therefore does not need to change the glue state. Due to the same reason the verification of the glue invariant is trivial.

On Figure 3.16 we can see the definitions of the methods *add* and *remove* of the class **Observer**. The pre- and post- conditions of the method *add* state that the method execution changes the glue state of **this** from $\texttt{st}_{Detached}$ to $\texttt{st}_{OnConstruction}$. The actual change of the state happens at the end of the method body. The pre-condition also requires the glue state of the input parameter o to be $\texttt{st}_{Attached}$. We need this pre-condition since the method can affect the glue state of o. Since we change the state of the object **this** from $\texttt{st}_{Detached}$ to $\texttt{st}_{OnConstruction}$ we have to verify that at the end of the method body the last two glue invariants hold. These invariants state that the doubly linked list formed by the observers is consistent with the `Sequence` stereotype slice. The invariant validity is guaranteed by consistent usage of heap field updates and stereotype operation calls. We also have to prove that the glue state of o is preserved by the method execution. It is also achieved by a consistent usage of heap field updates and stereotype operation calls.

In a similar way the pre- and post- conditions of the method *remove* state that the method execution changes the glue state of **this** from $\texttt{st}_{OnConstruction}$ to $\texttt{st}_{Detached}$. The actual change of the state happens at the end of the method body. The pre-condition also requires the glue state of ***this***.*nextObs* and ***this***.*prevObs* to be $\texttt{st}_{Attached}$. We need this pre-condition since the method can affect the glue states of these objects. Since we change the state of the object **this** from $\texttt{st}_{OnConstruction}$ to $\texttt{st}_{Detached}$ we have to verify that at the end of the method body the first two glue invariants hold. These invariants state that **this** forms singleton instances in the `Sequence` and the *ri* stereotype slice. This immediately follows from the post-condition of the call of stereotype operation `removeSeqElement`. We also have to prove that the glue states of ***this***.*nextObs* and ***this***.*prevObs* are preserved by the method execution.

On Figure 3.17 we can see the definition of the **Subject** glue states and glue invariants. The glue state of **Subject** is represented by the field `st` of type **SubjectState**. The values of the type **SubjectState** represent possible topological states of the subject:

- $\texttt{st}_{Head}$ - field *head* points to the head of the list of observers attached to the subject.
- $\texttt{st}_{NotHead}$ - field *head* points to some other object.

The **Subject** has three glue invariants. The first glue invariant states that if the subject glue state is $\texttt{st}_{Head}$ then the field *head* points to the head of the list of observers attached to the subject. The last two glue invariants establish the relation between **this**.*ri*.*Source* and *head*.`Sequence`.*Elements*.

this.*ri*.*Source* contains the set of observers which are attached to the subject. *head*.`Sequence`.*Elements* contains the set of observers which participate in the doubly-linked list to which *head* belongs. If an instance of the design pattern is not under construction then **this**.*ri*.*Source* has to be equal to *head*.`Sequence`.*Elements*. Nevertheless during the design pattern update it is possible that *head*.`Sequence`.*Elements* is already updated while **this**.*ri*.*Source* is not yet updated. In this case an observer can be attached to a doubly-linked list but not to the instance of the `RelationInversion` stereotype slice *ri*, where the subject participates. The glue state of these observers have to be $\mathtt{st}_{OnConstruction}$. The last two glue invariants state that **this**.*ri*.*Source* and *head*.`Sequence`.*Elements* contain the same elements excluding those whose glue state is $\mathtt{st}_{OnConstruction}$. If the *head* is **null** then the value of *head.ri.Elements* is undefined. Therefore in the glue invariants we use `seqElements`(*head*) instead of *head.ri.Elements*. `seqElements` is defined on Figure 3.14 and returns the empty set if *head*=**null**.

The constructor of **Subject** sets the glue state of the freshly allocated object to $\mathtt{st}_{Head}$. The validity of the glue invariant at the post state follows from the fact that the default value of *head* is **null** and that according to the post-condition of `createSink` **this**.*ri*.*Source* is equal to the empty set.

The method *updated* does not change the heap topology and therefore does not need to change the glue state. Due to the same reason verification of the glue invariant is trivial.

On Figure 3.18 we can see the definitions of the methods *attach* and *detach* of the class **Subject**. The method *attach* adds *observer* to the list of observers which are attached to the subject by calling of the method *add* of *observer*. The method execution violates the first glue invariant which states that *head* points to the head of the list of observers attached to the subject. Therefore before the method call of *add* we change the glue state of the subject to $\mathtt{st}_{NotHead}$. The glue state of the subject is restored to $\mathtt{st}_{Head}$ after the value of *head* is set to *observer* which is the new head of the list of observers.

The method *attach* also requires the state of the object *observer* to be $\mathtt{st}_{Detached}$. Execution of the method *add* changes the state of the object *observer* from $\mathtt{st}_{Detached}$ to $\mathtt{st}_{OnConstruction}$. At the end of the method we change the state of the object *observer* from $\mathtt{st}_{OnConstruction}$ to $\mathtt{st}_{Attached}$.

The method *detach* removes *observer* from the list of observers which are attached to the subject by calling of the method *remove* of *observer*. The method execution can violate the first glue invariant which states that *head* points to the head of the list of observers attached to the subject. Therefore before the method call of *remove* we change the glue state of the subject to $\mathtt{st}_{NotHead}$. The glue state of the subject is restored to $\mathtt{st}_{Head}$ after the value of *head* is set to the new head of the list of observers.

The method *detach* also ensures that after the method execution the state of the object *observer* is $\mathtt{st}_{Detached}$. According to the behavioral invariant `invSubject` which we consider in **Subsection 3.4.5** the state of *observer* before the method execution is $\texttt{st}_{Attached}$. At the beginning of the method we change the state of the object *observer* from $\texttt{st}_{Attached}$ to $\texttt{st}_{OnConstruction}$. Execution of the method *remove* changes the state of the object *observer* from $\texttt{st}_{OnConstruction}$ to $\texttt{st}_{Detached}$.

3.4.4 Stereotype operations for source code specification

In **Subsection 3.4.3** we have described how stereotypes can be used to specify ghost updates of stereotype items. If a method contains such a stereotype operation call then the precise specification of the method has to specify how the execution of the stereotype operation changes the values of the stereotype items. The same is true for loops. If a loop body calls a stereotype operation then the effect of the operation has to be specified in the loop invariant. As we have seen in **Subsection 3.4.3** a specification of a stereotype call can have a significant size. Therefore in this subsection we describe how stereotype operation calls can be used in the method specifications and the loop invariants.

Let us now consider method stereotype specifications. A method stereotype specification is located in the program source code between the method header and the method body. A stereotype method specification consists of the keywords `stereotype` followed by a stereotype operation call. The stereotype operation call can depend on the specified method input parameters and describes how the method execution changes the values of the stereotype items. Most of the methods affect the states of only a few stereotype slices and preserve the values of the stereotype items of all other stereotype slices. Therefore if a stereotype slice *StName* is not mentioned in the method stereotype specifications we add $\texttt{stereotype}\ \langle \mathit{StName} \rangle \texttt{skip}()$ to the method specifications.

We define the semantics of a method stereotype specification by reducing it to method pre- and post- conditions. In **Definition 19** we have described how for a given stereotype operation call of a stereotype operation **op** to construct the pre-condition $\mathbf{pre}'_{\mathbf{op}}$ and the post-condition $\mathbf{post}'_{\mathbf{op}}$ which specify the operation call. We use $\mathbf{pre}'_{\mathbf{op}}$ and $\mathbf{post}'_{\mathbf{op}}$ to define the semantics of the `stereotype` specification.

Definition 22 (Semantics of the method stereotype specification). *The method specification* $\texttt{stereotype}\langle \mathrm{StName}_1, \ldots, \mathrm{StName}_n \rangle$ $\boldsymbol{op}(exp_1, \ldots, exp_m)$ *is equivalent to the pair of method specifications* **`requires`** $\boldsymbol{pre}'_{\boldsymbol{op}}$ *and* **`ensures`** $\boldsymbol{post}'_{\boldsymbol{op}}$ *where the construction of* $\boldsymbol{pre}'_{\boldsymbol{op}}$ *and* $\boldsymbol{post}'_{\boldsymbol{op}}$ *is defined in* ***Definition 19***. *Here* $\mathrm{StName}_1, \ldots, \mathrm{StName}_n$ *refer to the states of stereotype slices before the method execution and* $\mathrm{StName}'_1, \ldots, \mathrm{StName}'_n$ *refer to the states of stereotype slices after the method execution.*

In the similar way we use stereotype operation calls to specify loop invariants. A stereotype loop specification consists of the keyword `stereotype` followed by a stereotype operation call. The stereotype operation call can depend on the method input parameters and local variables. The stereotype operation call describes how all executions of the loop body which were executed before the current iteration have changed the values of stereotype items.

Definition 23 (Semantics of stereotype loop specification)**.** *A stereotype loop specification* $\boldsymbol{stereotype}\langle \mathrm{StName}_1, \ldots, \mathrm{StName}_n\rangle \boldsymbol{op}(exp_1, \ldots, exp_m)$ *is equivalent to the loop invariant specification* `Loop Invariant` $\boldsymbol{post'_{op}}$ *where the construction of* $\boldsymbol{post'_{op}}$ *is defined in* ***Definition 19****. Here* $\mathrm{StName}_1, \ldots, \mathrm{StName}_n$ *refer to the states of stereotype slices before the whole loop execution and* $\mathrm{StName}'_1, \ldots, \mathrm{StName}'_n$ *refer to the states of stereotype slices before the current iteration of the loop.*

Let us now consider method specifications of the implementation of the observer design pattern.

On Figure 3.15 we can see the definition of the constructor and the method *update* of the class **Observer**. The constructor specification duplicates the call of the stereotype constructor `createObserver`. The body of the method *update* does not contain a call of a stereotype operation and therefore we do not need stereotype method specifications to specify it. The first two pre-conditions check that the input parameter *sub* is the subject to which the method receiver is attached. The next two specifications are related to behavioral invariants which we describe later.

On Figure 3.16 we can see the definitions of the methods *add* and *remove* of the class **Observer**. The body of the method *add* contains a call of the stereotype operation `addRelationInversion`. The operation is executed if the input parameter *o* is not equal to **null**. This behavior is equivalent to an `addSeqBegin` stereotype operation call. Therefore the only stereotype specification of the method *add* contains a call of `addSeqBegin`.

Method *remove* contains the only stereotype operation call which is duplicated in the method specification.

On Figure 3.17 we can see the definition of the constructor and *updated* method of the class **Subject**. Similarly to the class **Subject** the constructor specification duplicates the only stereotype operation which is executed by the constructor body. Method *updated* does not contain a call of a stereotype operation and therefore we do not need stereotype method specifications.

On Figure 3.18 we can see the definitions of the methods *attach* and *detach* of the class **Subject**. The specification of methods *attach* consists of two stereotype calls. One is the duplication of the `addRelationInversion` call which is executed in the method body. Another one is the call of `addSeqBegin` which is executed by the method *add* which is called from the *attach* method body. In a similar way the specification of the method *detach* consists of the call of the stereotype operation `removeRelationInversion` which is executed from the method body and from the call of the operation `removeSeqElement` which is executed by the method *remove* which is called from the body of the method *detach*.

We can see that most of the method stereotype specifications are a duplication of the stereotype operations calls from the bodies of the specified methods or of the operations which are called from methods which are called from the specified method. For non-recursive methods the stereotype method specifications can be inferred from the method bodies. As we have mentioned above, in this thesis we describe only the most essential machinery used in the stereotype-based verification and do not consider specification inference. Nevertheless we believe that the inference of stereotype method specifications for non-recursive methods is an interesting direction for the future work.

The stereotype method specifications of recursive methods have to specify the accumulated effect of the execution of an unbounded number of calls of the recursive methods. Inference of such specifications is similar to the inference of loop invariants and can hardly be done in the general case. In **Section 5** we consider specifications of several examples of methods which do heap updates in a recursive way.

3.4.5 Behavioral invariants

In the previous subsections we have described how stereotypes and glue invariants can be used to specify the heap topology of the verified program. In this section we consider how to specify the behavioral invariants.

Behavioral invariants use stereotype items and glue invariant states to establish the relations between the values of the fields of non-reference types. To keep our approach simple we do not introduce any particular methodology for the behavioral invariants. Instead we would like to introduce a predicate for each behavioral invariant and explicitly specify in pre-/post-conditions for which objects this predicate holds. On top of this framework any particular behavioral invariants methodology can be built. This methodology can use topological information provided by the stereotypes and the glue invariants and extra constraints to guarantee behavioral invariants preservation in a more systematic way and with less specification overhead. Here we do not consider any particular invariants methodologies. By this we try to avoid the introduction of extra machinery and keep our approach relatively simple.

An invariant can be formalized as a predicate Inv with a single input parameter of a reference type. The value of the predicate depends on the value of the input parameter and current state of the program heap. The invariant holds for an object o if and only if $Inv(o)$ holds. On the other hand if the invariant does not hold for the object o then we know nothing about it. It could be a problem in case we break an invariant in a method m and we are going to restore it later on in some other method m'. In this case we have to duplicate the parts of the invariant specification in specifications of all methods which are called between the execution of m and m'. This results in two problems: first of all we have a huge specification overhead, second of all we expose the invariant details to a method client and break information hiding. We address these problems by adding extra parameters to the invariant predicate. These extra parameters specify in an abstract way how the state of the object differs from one in which the invariant holds. In other words the extra parameter contains the information about the way in which the object state has to be changed in order to restore the invariant. We call this extra parameters *behaviorial invariant states*. This extension prevents the exposure of the invariant detail and the duplication of invariants specifications. Instead of breaking the invariant, the method specifies how it changes the behaviorial invariant states of the affected objects. Later on the information about the current behaviorial invariant state of an object can be used to restore its invariant.

Let us now consider the behavioral specifications of the implementation of the observer design pattern. Behavioral invariants of the implementation are provided in Figure 3.19.

`invObserver` is a behavioral invariant of the observer. `invObserver`(o) holds if and only if the value of the field *value* of o is equal to the value of the subject to which o is attached.

`invSubjectSt` is a behavioral invariant of the subject.
`invSubjectSt`(s, R) holds if and only if all the observers which are attached to the subject s and do not belong to the set R are synchronized with the subject. Here the value of the set R is the behavioral state of the subject. Set R describes which observer has to be updated in order to restore the invariant of the subject.

`invSubject` is another behavioral invariant of the subject.
`invAllAttached`(s) holds if and only if the glue states of all the observers which are attached to s are $\texttt{st}_{Attached}$.

`invSubject` is the behavioral invariant of the subject which combines all of the above. `invSubject`(s) holds if and only if `invAllAttached`(s), the behavioral state of the subject is $\varnothing$, and the glue state of the subject is $\texttt{st}_{Head}$.

Let us now consider how the implementation of the observer design pattern uses the behavioral invariants.

The constructor and the method *update* of the class **Observer** is provided on Figure 3.15. After the constructor execution the observer is not attached to a subject. Until the observer is not attached to a subject the value of the field *value* can be arbitrary. Therefore the constructor's specification does not mention any behavioral invariants.

The pre-condition of the method *update* requires validity of the `invAllAttached` invariant for the subject to which the method receiver is attached. The method post-condition guarantees that the method execution restores `invObserver` invariant of the method receiver. The constructor's specification does not mention any behavioral invariants.

Method *update* guarantees the preservation of the `invSubject` invariant. The assignment of a new value to the field *value* breaks the invariants of all attached observers. The loop restores the broken invariants. The specification of the loop invariants relies on the subject behaviorial invariant states. Invariants of all the attached observers are broken before the loop execution. Therefore the subject behavioral invariant state is equal to the set of the attached observers. Each loop iteration restores the invariant of one of the observers. The observer whose invariant is restored is removed from the subject invariant state. After the last iteration execution the subject invariant state is equal to the empty set. This implies that `invSubject` invariant is restored.

Methods *add* and *remove* of the class **Observer** are provided on Figure 3.16. Both of them preserve the `invSubject` invariant of the method receiver.

As we have mentioned above we would like to guarantee the preservation of the behavioral invariants for all instances of the design pattern. This is achieved via a precise frame properties specification. The frame properties guarantee that locations which are mentioned in the behavioral invariants of other instances of the design pattern are not affected by the methods of the verified implementation. This precise specification of the method frames is enabled by the usage of stereotype slices in class specifications and by the validity of the stereotype system invariants and the glue invariant in the methods' post states.

3.4.6 Summary

In this section we shortly summarize the stereotype-based verification approach which is introduced above.

A user begins verification from the definition of the stereotype slices which are used to specify the program. A stereotype slice is defined by mentioning it in the class header. If a class mentions the stereotype slice then we say that the class participates in the stereotype slice. It is possible that several classes participate in the same stereotype slice. In this case the stereotype slice identifies the semantic relation between class instances.

To bind the items of stereotype slices and heap fields the user has to introduce a glue invariant. Glue invariants have a visible state semantics. Its validity can be assumed at the beginning of the method body and after each method call. It has to be verified at the end of the method body and before each method call. The verification of the glue invariant validity is done by means of the **WPC**.

Then the methods of the verified classes have to be annotated with ghost updates. Ghost updates describe how the values of the items of a stereotype slice change during the program execution. We use stereotype operation calls to specify the ghost updates. The post-condition of each stereotype operation call can be assumed, but the pre-condition has to be verified. The verification of stereotype operation call pre-conditions is done by means of the **WPC**. According to **Theorem 1** we can also assume the validity of the stereotype system invariants of all the stereotype slices at the each program point.

Then the user has to provide method stereotype specifications. These specifications describe the accumulated effect on stereotype slices. They are described by means of stereotype operation calls. Each stereotype call is reduced to the usual method pre- and post- conditions which mentions the stereotype items. These pre- and post- conditions are verified with the rest of the method pre- and post- conditions by means of the **WPC**

The outcome of the above verification steps is the heap topology verification. At the top of the heap topology we specify the behavioral invariants. We specify the behavioral invariant as usual predicates which depend on the object for which the invariant is specified and the current state of the program heap. Later on we use the usual methods pre- and post- conditions to specify when and which behavioral invariants can be assumed, and when they have to be verified. Preservation of behavioral invariants of other design patterns and data structures is guaranteed by a precise specification of the frame properties.

Chapter 4

Stereotype examples

In the previous section we introduced the notion of stereotype and demonstrated its applications on the example of the `RelationInversion` stereotype. In this section we introduce more advanced stereotypes; `Sequence` and `Tree`. These stereotypes have a wide application in software specification and verification. Later on (see **Chapter 5**) we will demonstrate how these stereotypes can be used to verify the composite design pattern, list with iterator, and **PIP**.

4.1 `Sequence`

We begin the development of a stereotype from the identification of values of interest. For each of them we introduce a stereotype item. In the case of `Sequence` we are interested in the previous and the next value for each element of the sequence. To capture these notions we introduce the `prev` and `next` stereotype items of type *ref*. Also, for each sequence we are interested in the first and the last element of the sequence. Since their values are the same for all elements of a sequence we model them as `static` stereotype items `first` and `last`. On Figure 4.1 a stereotype instance of `Sequence` is depicted. The figure presents a view from the perspective of the object surrounded by the solid rectangle. Doted rounded rectangles depict values of stereotype items for this object.

On Figure 4.2 we present the definition of the `Sequence` stereotype. We express that `first` and `last` are not null by defining their types as *ref!*. We use the `next` stereotype item as a basis for the formalization of the semantics of the other stereotype items. The first invariant defines the semantics of the `prev` stereotype item. We define `prev` as the inversion of `next`. The second invariant defines the semantics of the `first` stereotype item. The first element of the sequence is the only one which doesn't have a predecessor. Since `first` is defined as non-null we know that the first element exists. The third invariant defines the semantics of `last` in a similar way. The

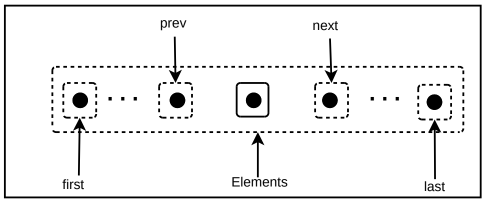


Figure 4.1: A stereotype slice of type `Sequence`. Here we denote objects as black circles. Dotted rounded rectangles depict values of stereotype items of the object which is surrounded by the solid rounded rectangle.

`Stereotype Sequence`{
`items`
    `static first`, `last`: *ref!* ;
    `next`, `prev`: *ref* ;

`invariants`
    (1) $\forall o, o' : o.\texttt{prev} = o' \Leftrightarrow o'.\texttt{next} = o$
    (2) $\forall o : o = \texttt{first} \Leftrightarrow o.\texttt{prev} = \textbf{null}$
    (3) $\forall o : o = \texttt{last} \Leftrightarrow o.\texttt{next} = \textbf{null}$

`instID`= `first`
}

Figure 4.2: `Sequence` stereotype.

last element of the sequence is the only one which doesn't have a successor. Since `last` is defined as non-null we know that the last element exists. The last part of the stereotype is the definition of the instance ID. Here we use the first element of a sequence as the instance ID. Each sequence instance has a first element, and first elements of different stereotype instances differ.

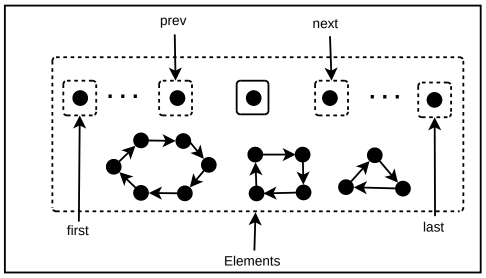


Figure 4.3: An example of an undesirable stereotype slice of type `Sequence`. Here an arrow between two objects denotes that they are in the "next" relation.

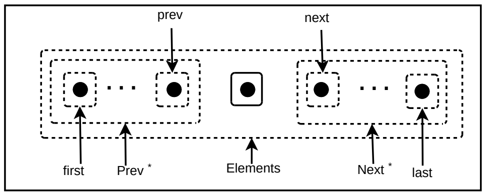


Figure 4.4: A stereotype slice of type Sequence.

Stereotype Sequence{
items
    static first, last: *ref!* ;
    next, prev: *ref* ;
    $\texttt{Next}^*$, $\texttt{Prev}^*$: *Reg* ;

invariants
    //Definitions of semantics of stereotype items
    (1) $\forall o, o' : o.\texttt{prev} = o' \Leftrightarrow o'.\texttt{next} = o$
    (2) $\forall o : o = \texttt{first} \Leftrightarrow o.\texttt{prev} = \textbf{null}$
    (3) $\forall o : o = \texttt{last} \Leftrightarrow o.\texttt{next} = \textbf{null}$
    (4) $\texttt{last}.\texttt{Next}^* = \varnothing$
    (5) $\forall o : o \neq \texttt{last} \Rightarrow o.\texttt{Next}^* = o.\texttt{next}.\texttt{Next}^* \cup \{o.\texttt{next}\}$
    (6) $\texttt{first}.\texttt{Prev}^* = \varnothing$
    (7) $\forall o : o \neq \texttt{first} \Rightarrow o.\texttt{Prev}^* = o.\texttt{prev}.\texttt{Prev}^* \cup \{o.\texttt{prev}\}$
    (8) $\forall o : \mathit{Elements} = o.\texttt{Prev}^* \cup \{o\} \cup o.\texttt{Next}^*$

    //Redundant invariants
    (9) $\forall o : o.\texttt{Prev}^* \sharp o.\texttt{Next}^*$
    (10) $\forall o : o \notin o.\texttt{Next}^*$
    (11) $\forall o : o \notin o.\texttt{Prev}^*$
    (12) $\forall o : o \neq \texttt{last} \Rightarrow \texttt{last} \in o.\texttt{Next}^*$
    (13) $\forall o : o \neq \texttt{first} \Rightarrow \texttt{first} \in o.\texttt{Prev}^*$
    (14) $\forall o, o' : o' \in o.\texttt{Next}^* \Rightarrow o'.\texttt{Next}^* \subseteq o.\texttt{Next}^*$
    (15) $\forall o, o' : o' \in o.\texttt{Prev}^* \Rightarrow o'.\texttt{Prev}^* \subseteq o.\texttt{Prev}^*$

instID= first
}

Figure 4.5: An updated version of the Sequence stereotype.

This stereotype definition captures the main values of interest and some of their properties. Nevertheless, it has significant disadvantages. First of all it is not precise enough. It means that there are stereotype slices which do not fit our intuition about sequences but satisfy the invariants of the

`Sequence` stereotype. An example of such a stereotype slice is presented on Figure 4.3. There we use an arrow between two objects to denote that they are in the ”next” relation. We can see that there are three cyclic structures at the bottom of the figure (hexagon, square, and triangle) which obviously contradict the idea of the sequence. On the other hand, it can be easily checked that this stereotype slice satisfies the invariants of the `Sequence` stereotype.

Another problem is that the proposed stereotype definition is not expressive enough. For instance, let us assume we have a loop which traverses a list and applies some transformations to the list elements. To express the loop invariant we need to specify sets of objects reachable by transitive closure of `prev` and `next`.

One way to address these issues is to use transitive closure. Since transitive closure is not feasible for automatic verification, we choose another approach. We introduce two additional stereotype items; `Next`* and `Prev`*. We use them to capture sets of objects reachable by transitive closure of `next` and `prev`, respectively. We call them right and left sequence tails. On Figure 4.4 a stereotype instance of the improved `Sequence` stereotype is depicted. We use these additional stereotype items to introduce additional stereotype invariants. The additional invariants provide a more precise sequence description. In addition `Next`* and `Prev`* can be used in specifications (e.g. loop invariants).

On Figure 4.5 we present an updated version of the `Sequence` stereotype. There are two kinds of stereotype invariants. Invariants (1) to (8) are definitions of the semantics of the stereotype items. Invariants (9) to (15) are redundant invariants which can be proven as corollaries of invariants (1) to (8). Nevertheless, most of these proofs require induction and therefore can be problematic for automatic verification. On the other hand proving that these invariants are preserved by the stereotype operations can be performed automatically. In such a way we avoid unnecessary interactive theorem proving.

Let us first consider the invariants which define the semantics of the stereotype items. The fist three stereotype invariants are the same as in the initial version of the stereotype. Invariants (4) and (5) define the semantics of the right tail. The right tail of the last element is empty set. The right tail any other element is the union of the next element and the tail of the next element. Invariants (6) and (7) define the semantics of the left tail in a similar way. Invariant (8) defines the semantics of the *Elements* of a sequence. The elements of a sequence is the union of the current element and the left and right tails. Altogether invariants (1) to (8) precisely define the semantics of sequences. A stereotype instance of `Sequence` satisfies invariants (1) to (8) if and only if the instance is a sequence.

Let us now consider invariants (9) to (15). As we mentioned above they are redundant and can be proven as corollaries of invariants (1) to (8).

```
addSequenceRelation⟨Sequence⟩($o : ref!, o' : ref!$)
{
Pre-conditions:
    $o$.instID $\neq o'$.instID ;
Input instances:
    $inInst_1 = o$ ;
    $inInst_2 = o'$ ;
Output instances:
    $outInst = inInst_1 \cup inInst_2$ ;
Transformations:
    $\langle o.\texttt{last}, \texttt{next}, o'.\texttt{first} \rangle$ ;
    $\langle o'.\texttt{first}, \texttt{prev}, o.\texttt{last} \rangle$ ;
    $\langle outInst, \texttt{first}, o.\texttt{first} \rangle$ ;
    $\langle outInst, \texttt{last}, o'.\texttt{last} \rangle$ ;
    $\langle inInst_1, \texttt{Next}^*, \mathbf{v}.\texttt{Next}^* \cup inInst_2 \rangle$ ;
    $\langle inInst_2, \texttt{Prev}^*, \mathbf{v}.\texttt{Prev}^* \cup inInst_1 \rangle$ ;
}

removeSequenceRelation⟨Sequence⟩($o : ref!$)
{
Local variables:
    $o' = o.\texttt{next}$ ;
Pre-conditions:
    $o \neq o\texttt{last}$ ;
Input instances:
    $inInst = o$ ;
Output instances:
    $outInst_1 = inInst \setminus o.\texttt{Next}^*$ ;
    $outInst_2 = o.\texttt{Next}^*$ ;
Transformations:
    $\langle o, \texttt{next}, \mathbf{null} \rangle$ ;
    $\langle o', \texttt{prev}, \mathbf{null} \rangle$ ;
    $\langle outInst_1, \texttt{last}, o \rangle$ ;
    $\langle outInst_2, \texttt{first}, o' \rangle$ ;
    $\langle outInst_1, \texttt{Next}^*, \mathbf{v}.\texttt{Next}^* \setminus outInst_2 \rangle$ ;
    $\langle outInst_2, \texttt{Prev}^*, \mathbf{v}.\texttt{Prev}^* \setminus outInst_1 \rangle$ ;
}
```

Figure 4.6: `addSequenceRelation` and `removeSequenceRelation` stereotype operations.

We introduce them to avoid inductive proofs. Invariants (9) - (11) state acyclicity of the sequence. Invariant (9) guarantees disjointness of the left

```
constructor createSingletonSequence⟨Sequence⟩($o$ : $ref!$)
{
        ⟨$o$, first, $o$⟩ ;
        ⟨$o$, last, $o$⟩ ;
        ⟨$o$, next, null⟩ ;
        ⟨$o$, Next$^*$, $\varnothing$⟩ ;
        ⟨$o$, prev, null⟩ ;
        ⟨$o$, Prev$^*$, $\varnothing$⟩ ;
}
```

Figure 4.7: Constructor of Sequence stereotype.

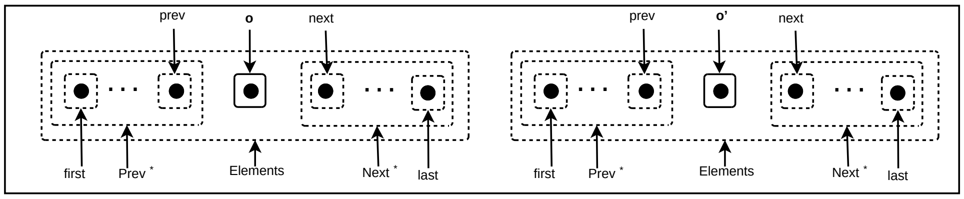
```
function Bool isSingletonSequence⟨Sequence⟩($o$ : $ref!$)
        $:= o.Elements = \{o\}$
```

Figure 4.8: A predicate which checks that a given sequence instance is a singleton.

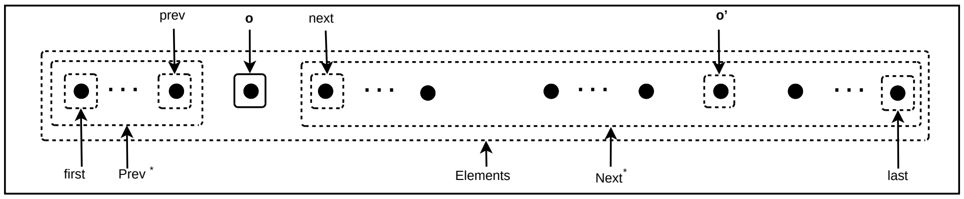
and the right tail. Here we denote disjointness of sets A and B as $A\sharp B$. Invariants (10) and (11) guarantee that an element does not belong to its right or left tail. Invariant (12) states that the last element belongs to right tails of all elements excluding the last element. This invariant guarantees that if we traverse a sequence via the `next` stereotype item we will eventually reach the last element. Invariant (13) guarantees the analogous property for the left tail and the first element. Invariant (14) states that the right tail of an object o is a superset of the right tail of each object o' in the right tail of o. This property is extremely useful for verification of loop invariants. Invariant (15) guarantees the same property for the left tail.

Now, we have a precise enough description of `Sequence` and we can introduce stereotype operations. We begin with the stereotype constructors. Since there is only one role which an element of a sequence can play, we need only one stereotype constructor. Namely, one which creates a singleton sequence. The definition of `createSingletonSequence` is presented on Figure 4.7. As you can see it has one input parameter. It is a reference to a freshly allocated object which participates in a stereotype slice of type `Sequence`. The constructor states that both tails of the sequence are empty, there are no previous and next elements, and the initialized object plays both roles of the first and of the last element of the sequence. An execution of the constructor results in the creation of the singleton sequence which consists of the object referred to by the input parameter. As for any other stereotype operation it can be proven that the constructor preserves the stereotype invariants which we introduced above.

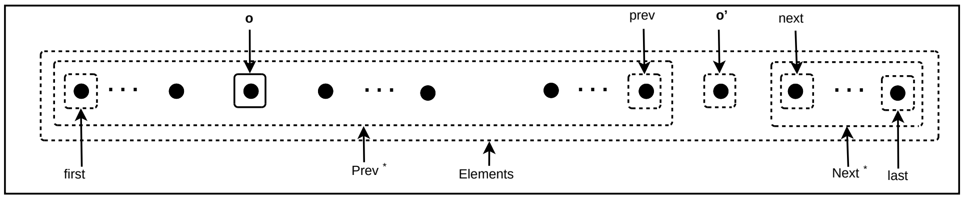
Since we have to deal with singleton sequence instances quite often we introduce a special predicate `isSingletonSequence` to characterize them. We define `isSingletonSequence` on Figure 4.8 as Boolean function. `isSingletonSequence` states that a sequence instance is a singleton if and only if its *Elements* is a singleton set. The rest of a singleton sequence properties (e.g. both tails are empty) can be inferred from this definition and the stereotype invariants.



(a) State of the stereotype slice before the statement `addSequenceRelation`(o, o').



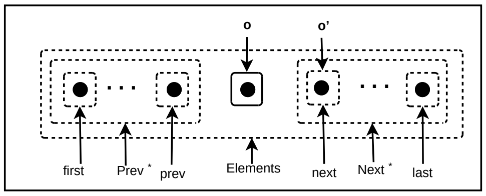
(b) State of the stereotype slice from the perspective of o after the statement `addSequenceRelation`(o, o').



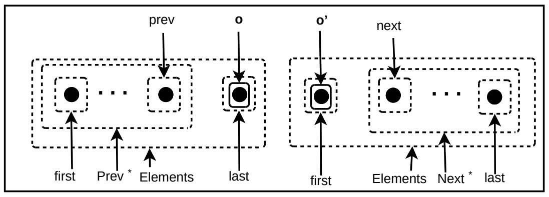
(c) State of the stereotype slice from the perspective of o' after the statement `addSequenceRelation`(o, o').

Figure 4.9: The figure depicts an execution of the statement `addSequenceRelation`(o, o').

As soon as stereotype instances are created we need operations to transform them. On Figure 4.6 you can see two such stereotype operations. The first one is `addSequenceRelation`. It merges two sequence stereotype instances into one by establishing the `next` relation between the last element of the first stereotype instance and the first element of the second stereotype instance. The second stereotype operation is `removeSequenceRelation`. It is the inverse operation to `addSequenceRelation`. `removeSequenceRelation` splits a stereotype instance in two independent stereotype instances by cutting the `next` relation between two consecutive elements of the stereotype instance. Let us consider in more details how these operations work.



(a) State of the stereotype slice before the statement execution.



(b) State of the stereotype slice after the statement execution.

Figure 4.10: The figure depicts an execution of the statement `removeSequenceRelation`(o).

As you can see on Figure 4.6 `addSequenceRelation` takes two input parameters of reference types. We use these references to characterize the stereotype instances which participate in the operation; $inInst_1$ and $inInst_2$. The pre-condition guarantees that these stereotype instances are different. You can see the initial state of the stereotype slice on Figure 4.9(a). The operation merges the two input stereotype instances into one stereotype instance $outInst$. The transformation section begins by establishing the relation between the last element of the first sequence and the first element of the second sequence. The first two lines of the transformations section establish `next` and `prev` relations between these elements. The next two lines define that the first element of the output instance is the first element of the first input instance and, in a symmetric way, the last element of the output instance is the last element of the second input instance. And finally, the last two lines define the values of the left and right tails of the output instance. In the fifth line we add the elements of the second input instance to the right tails of the elements of the first input instance. In a symmetric way in the last line we add the elements of the first input instance to the left tails of the elements of the second input instance. You can see the state of the stereotype slice after the operation from the perspective of o on Figure 4.9(b), and from the perspective of o' on Figure 4.9(c).

As we mentioned before `removeSequenceRelation` is the inverse operation to `addSequenceRelation`. It takes one input parameter of the reference type. We use this reference to characterize the stereotype instance which participates in the operation and to identify the removed relation. You can see the initial state of the stereotype slice Figure 4.10(a). The source of the removed relation is o. We introduce a new local variable o' to refer to the sink of the removed relation. The pre-condition of the operation states that the input parameter is not the last element of the input instance. In the output instances section we define two output instances. The second one contains the right tail of o. The first one contains the rest of the elements of the input instance. The transformation section is the inverse to the transformation section of `addSequenceRelation` operation. It removes `next` and `prev` relations between o and o', and updates the other stereotype items correspondingly. You can see the state of the stereotype slice after the operation execution on Figure 4.10(b).

```
function ref cyclicNext⟨Sequence⟩(o : ref!) :=
     if (o = o.last) then o.first else o.next

function ref cyclicPrev⟨Sequence⟩(o : ref!) :=
     if (o = o.first) then o.last else o.prev
```

Figure 4.11: Next and previous elements of a cyclic list.

We use the `Sequence` stereotype to specify and verify various data structures and design patterns in **Chapter 5**. To facilitate usage of the stereotype to specify a specific structure we introduce additional functions. For instance, we can use the `Sequence` stereotype to specify a cyclic list. To do that we introduce two additional functions: `cyclicNext` and `cyclicPrev`. We present their definitions on Figure 4.11. A next element in a cyclic list is equal to the next element of the sequence if the current element is not the last in the sequence, otherwise it is equal to the first element of the sequence. We define `cyclicPrev` in a similar way. By adding these two simple functions we enable the application of the `Sequence` stereotype to verify yet another type of heap structure.

As mentioned in **Section 3.3** it is strongly desirable to construct for a stereotype a *universal transformation*; a stereotype operation which can be used to define an arbitrary transformation. In **Appendix A** we use `addSequenceRelation` and `removeSequenceRelation` to construct the most general operation for sequences transformation. Here we briefly describe how it is done.

The construction of universal transformations is discussed in **Subsection 6.5**. The main conclusion of the subsection is that for some stereotypes the construction of a universal transformation can be reduced to the construction of two operations. One should merge a set of stereotype instances in to a single stereotype instance. The second one is the inversion of the first one. It has to split a stereotype instance into a set of stereotype instances. Sequence is one of the stereotypes for which the above reduction can be used.

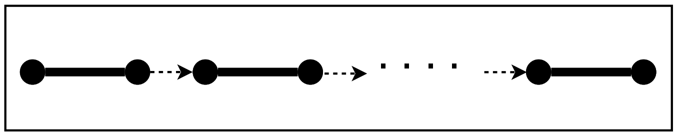


Figure 4.12: The figure depicts the state of the slice after an execution of the `addSetSequenceRelation` operation. Here we denote sequence instances as solid bold lines. Black circles depict the first and the last elements of a sequence instance. Dotted directed lines denote freshly added relations.

The only way to merge two sequence instances is to add a relation between the last element of the first one and the fist element of the second one. Because of this the only way how we can merge a set of sequences instances into a single sequence instance is the following one. First we chose an order in which we are going to merge the sequence instances and then according to the order we add relations between the last and the first elements of the sequences. The merging of a set of sequence instances is shown on Figure 4.12. We call the stereotype operation which merges a set of sequence instances `addSetSequenceRelation`. The signature of the operation is shown on Figure 4.13. The complete definition of the operation is provided in **Appendix A.1**.

The input parameters of the operation are the following:

- Ib is the instance identifier of the first sequence.
- Ie is the instance identifier of the last sequence.
- $Inst$ is a set of instances identifiers of merged sequences.
- $nextInst$ is an order over sequence instances. It maps a sequence identifier to a sequence identifier of the next sequence according to the order in which they are merged.
- $ElUnion$ is a redundant argument which maps an instance identifier o to the union of all elements of all sequence instance which are suc-

`addSetSequenceRelation`$\langle$`Sequence`$\rangle(Ib : ref!, Ie : ref!, Inst : Reg!,$
    $nextInst : ref \rightarrow ref, ElUnion : ref \rightarrow TReg)$

`removeSetSequenceRelation`$\langle$`Sequence`$\rangle$
    $(Ib : ref!, Ie : ref!, Inst : Reg!, nextInst : ref \rightarrow ref)$

Figure 4.13: Signatures of `addSetSequenceRelation` and `removeSetSequenceRelation` stereotype operations.

```
reverseSequence⟨Sequence⟩($l$ : $ref!$)
Local variables:
    $Ib = \texttt{old}(l.\texttt{first})$ ;
    $Ie = \texttt{old}(l.\texttt{last})$ ;
    $Inst = \texttt{old}(l.Elements)$ ;
    $\forall o : nextInst[o] = \texttt{old}(o.\texttt{next})$ ;
    $\forall o : nextInstRev[o] = \texttt{old}(o.\texttt{prev})$ ;
    $\forall o : ElUnion[o] = \texttt{old}(o.\texttt{Prev}^* \cup \{o\})$ ;
{
    removeSetSequenceRelation
        $(l.\texttt{first}, l.\texttt{last}, l.Elements \setminus \{\texttt{last}\}, nextInst)$ ;
    addSetSequenceRelation$(Ib, Ie, Inst, nextInstRev, ElUnion)$ ;
}
```

Figure 4.14: `reverseSequence` stereotype operation.

ceeded or equal to o according to the order $nextInst$. The parameter is redundant in a sense that it can be constructed from other parameters, but we require it to avoid usage of a transitive closure and to facilitate the proof.

Let us now consider the inversion of `addSetSequenceRelation`; `removeSetSequenceRelation` operation. It splits a single input sequence instance on a set of sequence instances. The signature of the operation is shown on Figure 4.13. The complete definition of the operation is provided in **Appendix A.2**.

The input parameters of the operation are the following:

- $Inst$ is a set of cut-points. It contains the initial points of `next` relations which are removed by the operation execution.
- $nextInst$ is a redundant parameter which for each cut-point provides the next cut-point according to the order which is induced by the `next` stereotype item. In other words, for each o from $Inst \setminus \{Ie\}$ $nextInst[o]$ is reachable from o by `next` and there is no other element from $Inst$ between them.
- Ib and Ie are the first and the last cut-points according to the order provided by $nextInst$.

Let us now consider an application of the sequence universal transformation. On Figure 4.14 we specify sequence reversal. The transformation consists of two calls. The first one splits the input sequence into a set of singleton sequences. The second one merges the singleton sequences in the reversed order. The local variable $nextInst$ contains the value of the input parameter $nextInst$ of the operation `removeSetSequenceRelation`. We use the values of the stereotype item `next` to specify the value of $nextInst$. The local variable $ElUnion$ contains the value of the input parameter $ElUnion$ of the operation `removeSetSequenceRelation`. We use the values of the stereotype item `Prev`* to specify the value of $nextInst$.

4.2 Tree

On Figure 4.15 we present the definition of the `Tree` stereotype. Similarly to the `Sequence` stereotype we begin the development of the `Tree` stereotype by identifying values of interest:

- The main value of the interest for the `Tree` stereotype is the parent of a node. We formalize it as stereotype item `parent` of type *ref*. Using the parent relation we can define all other values of interest.
- The root of a tree can be defined as a node of the tree which doesn't have a parent. We denote the root of a tree as a stereotype item `root`. Since any tree has a single root we declare it as a `static` stereotype item of type *ref*.
- The inversion of the parent relation produces the children of a node. We denote them by the stereotype item `Child`. Since a node can have an unbounded number of children the type of `Child` is *Reg*.
- The nodes of a tree reachable via the transitive closure of the parent relation are called ancestors. We denote them by the stereotype item `Anc` of type *Reg*.
- The nodes of a tree reachable via the transitive closure of the children are relation called descendants. We denote them by the stereotype item `Desc` of type *Reg*.
- To facilitate the verification of stereotype invariants we introduce the stereotype item $f_{\texttt{Desc}}$ of type *ref* $\rightarrow$ *ref*. The semantics of $f_{\texttt{Desc}}$ can be described by the following property: for each p and l such that l is a descendant but not a child of p, the value of $p.f_{\texttt{Desc}}[l]$ is both a child of p and an ancestor of l. We can think about $p.f_{\texttt{Desc}}[l]$ as an intermediate node between p and l. Later on we demonstrate how to use $f_{\texttt{Desc}}$ for the verification of stereotype invariants.

On Figure 4.16 a stereotype instance of `Tree` is depicted. Here we use the same notations that we used in the previous subsection.

Let us first consider the invariants which define the semantics of the stereotype items. Invariant (1) defines the semantics of the root. The root of a tree is a node of the tree which doesn't have a parent. Invariant (2)

```
Stereotype Tree{
items
    static root: $ref!$ ;
    parent: $ref$ ;
    Child, Anc, Desc: $Reg$ ;
    $f_{\texttt{Desc}}$: $Reg \to Reg$ ;

invariants
    //Definitions of the semantics of stereotype items
    (1) $\forall o : o = \texttt{root} \Leftrightarrow o.\texttt{parent} = \textbf{null}$
    (2) $\forall o, o' : o' \in o.\texttt{Child} \Leftrightarrow o'.\texttt{parent} = o$
    (3) $\texttt{root.Anc} = \varnothing$
    (4) $\forall o : o \neq \texttt{root} \Rightarrow o.\texttt{Anc} = o.\texttt{parent.Anc} \cup \{o.\texttt{parent}\}$
    (5) $\forall o, o' : o' \in o.\texttt{Desc} \Leftrightarrow o \in o'.\texttt{Anc}$
    (6) $\forall o : o.\mathit{Elements} = \texttt{root.Desc} \cup \{\texttt{root}\}$

    //Tree invariant
    (7) $\forall o, o' : (o \neq o' \wedge o' \notin o.\texttt{Desc} \wedge o \notin o'.\texttt{Desc}) \Rightarrow o'.\texttt{Desc} \sharp o.\texttt{Desc}$

    //Redundant invariants
    (8) $\forall o : o \neq \texttt{root} \Rightarrow \texttt{root} \in o.\texttt{Anc}$
    (9) $\forall o, o' : o \neq o' \Rightarrow o.\texttt{Child} \sharp o'.\texttt{Child}$
    (10) $\forall o : o.\texttt{Child} \subseteq o.\texttt{Desc}$
    (11) $\forall o : o \notin o.\texttt{Desc}$
    (12) $\forall o : o \notin o.\texttt{Anc}$
    (13) $\forall o, o' : o' \in o.\texttt{Desc} \Rightarrow o.\texttt{Child} \sharp o'.\texttt{Desc}$
    (14) $\forall o, o' : o' \notin o.\texttt{Desc} \Rightarrow o'.\texttt{Anc} \sharp o.\texttt{Desc} \cup \{o\}$
    (15) $\forall o, o' : o' \in o.\texttt{Desc} \Rightarrow o'.\texttt{Anc} \setminus o.\texttt{Desc} = o.\texttt{Anc} \cup \{o\}$
    (16) $\forall o, o' : o' \in o.\texttt{Desc} \Rightarrow o'.\texttt{Desc} \subseteq o.\texttt{Desc}$
    (17) $\forall o, o' : o' \in o.\texttt{Anc} \Rightarrow o'.\texttt{Anc} \subseteq o.\texttt{Anc}$

    //Definition of the semantics of the auxiliary
    //stereotype item
    (18) $\forall o : o.\texttt{Desc} \setminus o.\texttt{Child} = \bigcup_{c \in o.\texttt{Child}}^{o.f_{\texttt{Desc}}} c.\texttt{Desc}$

instID= root
}
```

Figure 4.15: Tree stereotype.

defines the semantics of the children. An element of a tree o' is a child of

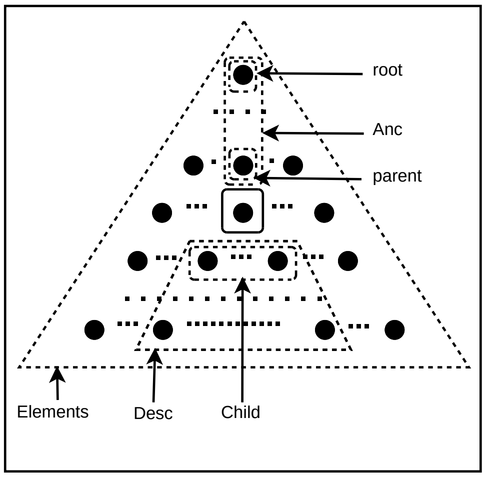


Figure 4.16: A stereotype slice of type the `Tree`.

```
constructor createSingletonTree⟨Tree⟩(o : ref!)
{
        ⟨o, root, o⟩ ;
        ⟨o, parent, null⟩ ;
        ⟨o, Child, ∅⟩ ;
        ⟨o, Anc, ∅⟩ ;
        ⟨o, Desc, ∅⟩ ;
}
```

Figure 4.17: Constructor of `Tree` stereotype.

another element o of the tree if and only if o is the parent of o'. Invariants (3) and (4) define the semantics of ancestors. Ancestors of root is the empty set. Ancestors of a non-root element is the union of the parent and the ancestors of the parent. Invariant (5) defines the semantics of the descendants. An element of a tree o' is a descendant of another element o of the tree if and only if o is an ancestor of o'. Invariant (6) defines the semantics of *Elements* of a tree. The elements of a tree is the union of the root element and the descendants of the root element.

Altogether invariants (1) to (6) precisely define the semantics of a DAG (directed acyclic graph). There is a structure which satisfies invariants (1) to (6) but which is not a tree. To filter out such properties we add invariant (7). The invariant guarantees disjointness of descendants of two different elements of a tree if neither of these two elements is a descendant of the other. Invariants (1) to (7) precisely define the semantics of trees.

Let us know consider invariants (8) to (17). They are redundant and can be proven as corollaries of invariants (1) - (7). Invariant (8) states that the

```
addTreeRelation⟨Tree⟩($o$ : $ref!$, $o'$ : $ref!$)
{
Pre-conditions:
    $o.\mathtt{instID} \neq o'.\mathtt{instID}$;
Input instances:
    $inInst_1 = o$;
    $inInst_2 = o'$;
Output instances:
    $outInst = inInst_1 \cup inInst_2$;
Transformations:
    $\langle o'.\mathtt{root}, \mathtt{parent}, o\rangle$;
    $\langle o, \mathtt{Child}, o.\mathtt{Child} \cup \{o'\}\rangle$;
    $\langle outInst, \mathtt{root}, o.\mathtt{root}\rangle$;
    $\langle o.\mathtt{Anc} \cup \{o\}, \mathtt{Desc}, \mathbf{v}.\mathtt{Desc} \cup inInst_2\rangle$;
    $\langle inInst_2, \mathtt{Anc}, \mathbf{v}.\mathtt{Anc} \cup o.\mathtt{Anc} \cup \{o\}\rangle$;
    $\langle o.\mathtt{Anc} \times inInst_2, f_{\mathtt{Desc}}, o.f_{\mathtt{Desc}}[\mathbf{v'}]\rangle$;
    $\langle o \times inInst_2, f_{\mathtt{Desc}}, o'\rangle$;
}

removeTreeRelation⟨Tree⟩($o$ : $ref!$)
{
Local variables:
    $o' = o.\mathtt{parent}$;
Pre-conditions:
    $o \neq o.\mathtt{root}$;
Input instances:
    $inInst = o$;
Output instances:
    $outInst_1 = inInst \setminus (o.\mathtt{Desc} \cup \{o\})$;
    $outInst_2 = o.\mathtt{Desc} \cup \{o\}$;
Transformations:
    $\langle o, \mathtt{parent}, \mathbf{null}\rangle$;
    $\langle o', \mathtt{Child}, o'.\mathtt{Child} \setminus \{o\}\rangle$;
    $\langle outInst_2, \mathtt{root}, o\rangle$;
    $\langle o.\mathtt{Anc}, \mathtt{Desc}, \mathbf{v}.\mathtt{Desc} \setminus outInst_2\rangle$;
    $\langle outInst_2, \mathtt{Anc}, \mathbf{v}.\mathtt{Anc} \setminus o.\mathtt{Anc}\rangle$;
}
```

Figure 4.18: The `addTreeRelation` and `removeTreeRelation` stereotype operations.

root belongs to the ancestors of all elements excluding the root. This invariant guarantees that if we traverse a tree via `parent` we will eventually reach

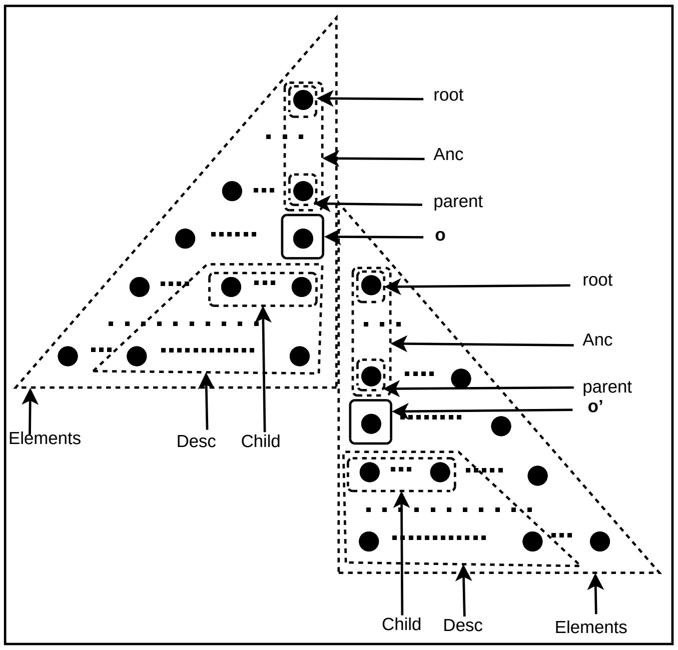


Figure 4.19: The figure depicts the state of the stereotype slice before execution of the statement `addTreeRelation`(o, o').

the root. Invariant (9) states that children of disjoint elements are disjoint. Invariant (10) states that children of an element are also descendants of this element. Invariants (11) and(12) state that an element is disjoint from its ancestors and descendants. Invariant (13) guarantees that descendants of a descendant of an element are disjoint from children of the same element. Invariant (14) states that if an element o' is not a descendant of an element o then the ancestors of o' and the union of o and descendants of o are disjoint. From the special case of this invariant when $o = o'$ we conclude that ancestors and descendants of an element are disjoint. Invariant (15) states that for each descendant o' of an element o, the ancestors of o' consist of three disjoint parts: ancestors of o, o, and some subset of descendants of o. Invariant (16) states that the descendants of a descendant of an element are also descendants of the element. In other words it states transitivity of the descendants relation. This property is extremely useful for the verification of loop invariants. Invariant (17) states the same property about ancestors.

Invariant (18) states that the difference between descendants and children of a node is equal to the union of the descendants of the node's children. This property is crucial for the specification of recursive updates of a tree. On the other hand the specification of a union of a set of sets requires usage of a quantifier alternation. One possible way to avoid a quantifier alternation is to provide explicit values for the existential quantifier. These values

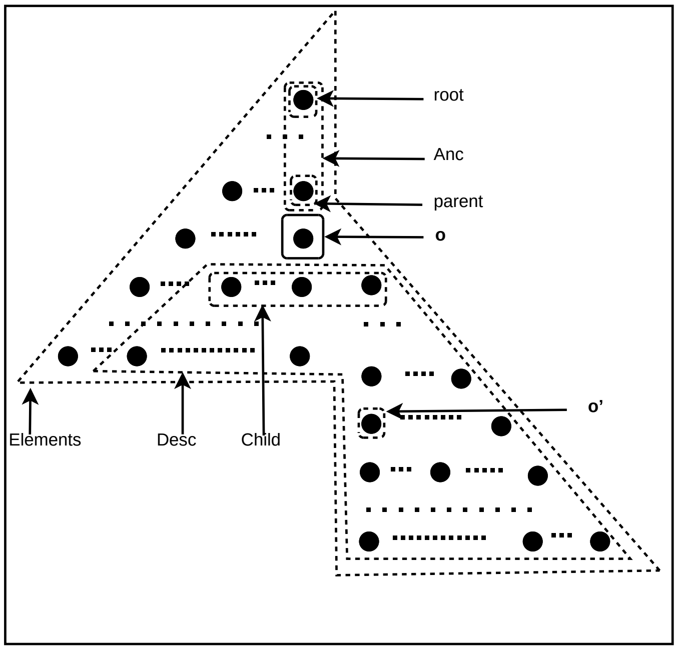


Figure 4.20: The figure depicts the state of the stereotype slice after execution of the statement `addTreeRelation`(o, o') from the perspective of o.

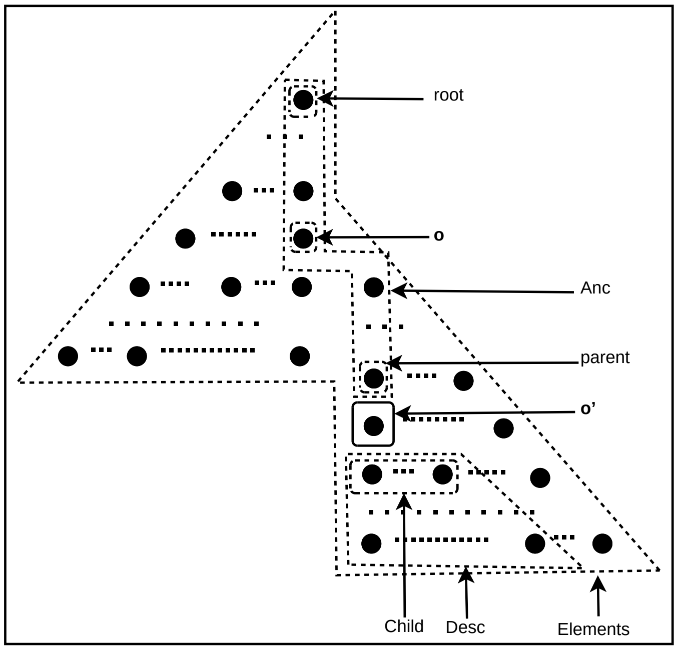
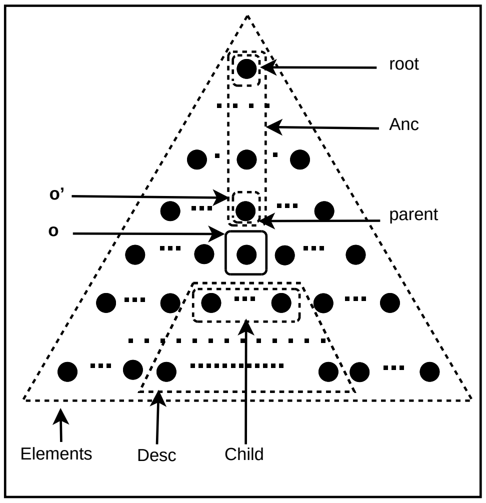
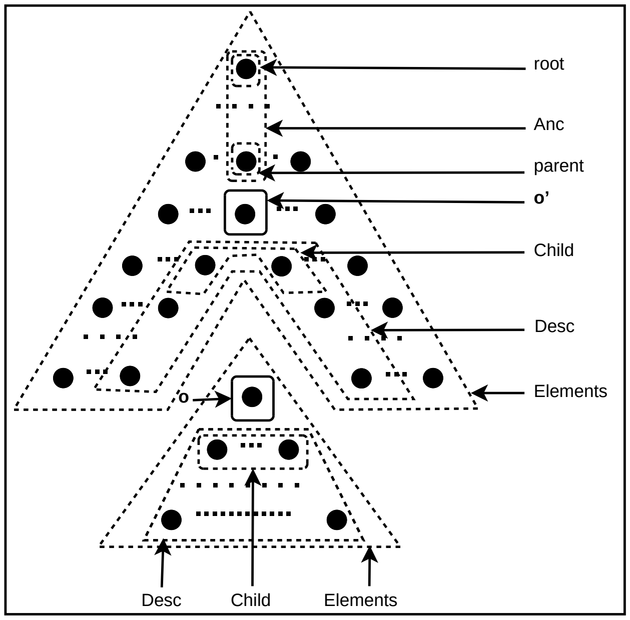


Figure 4.21: The figure depicts the state of the stereotype slice after the execution of the statement `addTreeRelation`(o, o') from perspective of o'.



(a) State of the stereotype slice before the statement execution.



(b) State of the stereotype slice after the statement execution.

Figure 4.22: The figure depicts the execution of the statement `removeTreeRelation`(o).

play the role of the hint for the theorem prover. In the invariant (18) we use superscript $f_{\texttt{Desc}}$ to identify that we are using $f_{\texttt{Desc}}$ as a hint for a theorem prover to prove the equality. We consider auxiliary proofs and details of the verification of hint functions in **Chapter 6**.

The last part of the stereotype definition is instance ID. Here we use the root of a tree as the instance ID. Each tree instance has a root and the roots of different stereotype instances are different.

Now, that we have the definition of `Tree` we can introduce stereotype operations. We begin with stereotype constructors. Since there is only one role which an element of a tree can play, we need only one stereotype constructor. Namely, one which creates a singleton tree. The definition of `createSingletonTree` is presented on Figure 4.17. It has one input parameter. It is a reference to a freshly allocated object which participates in a stereotype slice of type `Tree`. The constructor states that the element is the root, the element doesn't have a parent, and children, ancestors and descendants are empty. An execution of the constructor results in the creation of the singleton tree which consists of the object referred to by the input parameter. As for any other stereotype operation it can be proven that the constructor preserves the stereotype invariants which we introduced above.

`function Bool isSingletonTree`$\langle$`Tree`$\rangle(o : \mathit{ref!}) := o.\mathit{Elements} = \{o\}$

Figure 4.23: A predicate which checks that a given tree instance is a singleton.

Since we have to deal with singleton sequence instances quite often we introduce a special predicate `isSingletonTree` to characterize them. We define `isSingletonTree` on Figure 4.23 as Boolean function.
`isSingletonTree` states that a tree instance is a singleton if and only if its elements is a singleton set.

As soon as stereotype instances are created we need operations to transform them. On Figure 4.18 you can see two such stereotype operations. The first one is `addTreeRelation`. It merges two tree stereotype instances into one by establishing a `parent` relation between an element of the first stereotype instance and the root element of the second stereotype instance. The second stereotype operation is `removeTreeRelation`. It is the inverse operation to `addTreeRelation`. `removeTreeRelation` splits a stereotype instance in two independent stereotype instances by cutting a `parent` relation between two elements of the stereotype instance. Let us consider in more details how these operations work.

As you can see on Figure 4.18 `addTreeRelation` takes two input parameters of the reference type. We use these references to characterize the stereotype instances which participate in the operation; $inInst_1$ and $inInst_2$. The pre-condition guarantees that these stereotype instances are different. You can see the initial state of the stereotype slice on Figure 4.19. The operation merges two input stereotype instances into one stereotype instance $outInst$. The transformation section begins by establishing the relation between the root element of the second tree and the first input parameter. The first two lines of the transformations section establish the `parent` and `Child` relations between these elements. The next line defines that the root element of the output instance is the root element of the first input instance. The next two lines defines values of the descendants and ancestors of the output instance. In the fourth line we add the elements of the second input instance to the descendants of the elements which are either o or ancestors of o. In a symmetric way in the fifth line we add o and ancestors of o to the ancestors of the elements of the second input instance.

The last two lines describe how $f_{\texttt{Desc}}$ changes. Since $f_{\texttt{Desc}}$ is a function but not a single value, we use the following syntax to describe its updates; $\langle S_1 \times S_2, f_{it}, val\rangle$. Here f_{it} is the name of a stereotype item of a functional type, S_1 is a set of objects whose values of f_{it} is modified, S_2 is a set of parameters of function f_{it} for which the value of the function is changed, and val is an expression which denotes the new value of the function. val can depend on an old value of the function. To express such kinds of transformations we assume that S_1, S_2, and val use the predefined variables **v** and **v'** to refer to an object and input parameter for which value of $\mathbf{v}.f_{it}[\mathbf{v'}]$ is changed.

Let us now consider the sixth and the seventh line of `addTreeRelation`'s transformation section in more details. Since o and its ancestors have got new descendants their values of $f_{\texttt{Desc}}$ have to be extended to cover these new descendants. We can see that all paths from an element of the second input instance to an ancestor of o goes via o. The sixth line exploits this observation and states that for an ancestor of o the value of $f_{\texttt{Desc}}[\mathbf{v'}]$ is equal to value of $f_{\texttt{Desc}}[\mathbf{v'}]$ of o. We can see that all paths from an element of the second input instance to o goes via o'. On other hand o' is a child of o. The seventh line exploits these observations and states that the value of $f_{\texttt{Desc}}[\mathbf{v'}]$ of o is equal to o'.

You can see the state of the stereotype slice after the operation execution from the perspective of o on Figure 4.20, and from the perspective of o' on Figure 4.21.

As we mentioned before `removeTreeRelation` is the inverse operation to `addTreeRelation`. It takes one input parameter of the reference type. We use this reference to characterize the stereotype instance which participates in the operation and to identify the removed relation. You can see the initial state of the stereotype slice on Figure 4.22(a). The source of the removed relation is o. We introduce a new local variable o' to refer to the sink of the removed relation. The pre-condition of the operation states that the input parameter is not the root element of the input instance. In the output instances section we define two output instances. The second one contains o and the descendants of o. The first one contains the rest of the elements of the input instance. The transformation section is the inverse of the transformation section of the `addTreeRelation` operation. It removes the `parent` and `Child` relations between o and o', and updates

$\texttt{addSetTreeRelation1}\langle\texttt{Tree}\rangle(l : \mathit{ref!}, P : \mathit{Reg!})$

$\texttt{removeSetTreeRelation1}\langle\texttt{Tree}\rangle(P : \mathit{Reg!})$

$\texttt{addSetTreeRelation2}\langle\texttt{Tree}\rangle$
$(r_0 : \mathit{ref!}, R : \mathit{Reg!}, P : \mathit{Reg!}, r2P : \mathit{ref} \to TReg, p2r : \mathit{ref} \to \mathit{ref},$
$r2R : \mathit{ref} \to \mathit{Reg}, c2r : \mathit{ref} \to \mathit{ref})$

$\texttt{removeSetTreeRelation2}\langle\texttt{Tree}\rangle(r_0 : \mathit{ref!}, R : \mathit{Reg!}, P : \mathit{Reg!},$
$r2R : \mathit{ref} \to \mathit{Reg}, c2r : \mathit{ref} \to \mathit{ref}, f_{PEL} : \mathit{ref} \to \mathit{ref})$

$\texttt{addSetTreeRelation3}\langle\texttt{Tree}\rangle(r_0 : \mathit{ref} \to \mathit{ref!}, R_{All} : \mathit{Reg!}, P : \mathit{ref} \to \mathit{Reg!},$
$r2P : \mathit{ref} \to TReg, p2r : \mathit{ref} \to \mathit{ref}, r2R : \mathit{ref} \to \mathit{Reg}, c2r : \mathit{ref} \to \mathit{ref},$
$T : \mathit{Reg!}, t_0 : \mathit{ref}, tD : \mathit{ref} \to \mathit{Reg!}, f_{tD} : \mathit{ref}^2 \to \mathit{ref})$

$\texttt{removeSetTreeRelation3}\langle\texttt{Tree}\rangle(p_0 : \mathit{ref!}, p2r_0 : \mathit{ref} \to \mathit{ref}, R : \mathit{Reg!},$
$P : \mathit{Reg!}, r2R : \mathit{ref} \to \mathit{Reg}, c2r : \mathit{ref} \to \mathit{ref}, f_{PEL} : \mathit{ref}^2 \to \mathit{ref},$
$p2P : \mathit{ref} \to \mathit{Reg!})$

Figure 4.24: Signatures of stereotype operations from which the universal tree transformation is constructed.

other stereotype items correspondingly.

The only stereotype item whose value remains unchanged is $f_{\texttt{Desc}}$. The reason for this is that we only remove descendants of all participating objects. Before the operation execution $f_{\texttt{Desc}}$ is properly defined on $inInst \times inInst$, here the first set represents a receiver and the second a function parameter. After the operation execution we care only about values of the function on $outInst_1 \times outInst_1$ and on $outInst_2 \times outInst_2$. Correctness of the function on these sets is preserved by the operation. On the other hand we do not care about correctness of the function on $outInst_1 \times outInst_2$ and on $outInst_2 \times outInst_1$. These two observations explain why we can leave the value of $f_{\texttt{Desc}}$ unchanged.

You can see the the stereotype slice after the operation execution on Figure 4.22(b).

Let us now consider the construction of the universal transformation for the tree stereotype. The construction is quite complex; therefore here we provide only a brief description. The complete definition is provided in **Appendix B**.

The universal transformation of a tree consists of three levels. A level consists of two operations: one for the addition and one for the removal of relations. These operations are dual, one is an inversion of the other. On each level we use operations from previous levels. For each operation we

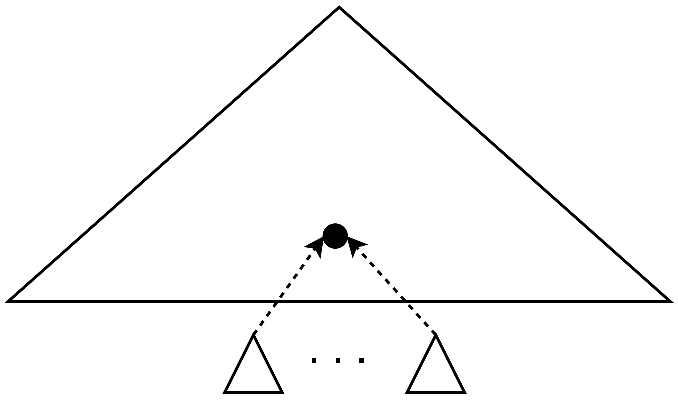


Figure 4.25: The figure depicts the state of the stereotype slice after an execution of the `addSetTreeRelation1` operation. Here we denote trees as triangles. Dotted directed lines denote freshly added relations. The black circle depicts an element of the tree to which we add new relations.

provide a detailed description by considering each element of the operation. For each level we also provide a figure which visualizes the result of an execution of an addition operation. We do not provide an analogous figure for removal operation. Since additional and removal operations are dual a figure for one of the operations also can be used as an explanation for the other operation.

The first level is defined in **Appendix B.1**. It consists of `addSetTreeRelation1` which is defined in **Appendix B.1.1** and `removeSetTreeRelation1` which is defined in **Appendix B.1.2**.

`addSetTreeRelation1` is the extension of `addTreeRelation` to the case when we add an unbounded number of sub-trees to an element of a tree. The operation is defined on Figure B.1. The result of the `addTreeRelation` execution is shown on Figure 4.25. `removeTreeRelation` is dual to `addSetTreeRelation1`. It removes a set of sub-trees which have the same parent. We can think about the operation as a generalization of `removeTreeRelation`.

Input parameters of `addTreeRelation` are:

- l: an element of the tree to which sub-tress are added.
- P: set of instances identifiers of added sub-trees.

The Input parameter of `removeTreeRelation` is

- P: set of roots of removed sub-trees.

The second level is defined in **Appendix B.2**. It consists of `addSetTreeRelation2` which is defined in **Appendix B.2.1** and `removeSetTreeRelation2` which is defined in **Appendix B.2.2**.

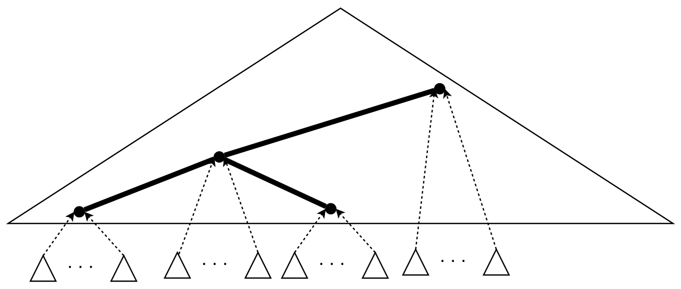


Figure 4.26: The figure depicts the state of the stereotype slice after an execution of the `addSetTreeRelation2` operation. Here we denote trees as triangles. Dotted directed lines denote freshly added relations. Black circles depict elements of the auxiliary tree relation. Solid lines between black circles denote an auxiliary tree relation.

`addSetTreeRelation2` is an extension of `addSetTreeRelation1` to the case
when we add sub-trees to an unbounded number of elements of a tree. A result of the operation execution is depicted on Figure 4.26. To make the operation feasible we require a specification developer to provide an auxiliary tree relation. There are two kinds of elements of the auxiliary tree relation. The first one is elements of the tree to which sub-trees are added. The second one is join points of the auxiliary tree relation. o is a join point of of the auxiliary tree relation if and only if there are elements of the auxiliary tree relation o_1 and o_2 such that o_1 and o_2 are descendants of o, o_1 and o_2 belong to different sub-trees of o, and there are no other elements of the auxiliary tree relation between o and o_1, and between o and o_2. On Figure 4.26 we denote elements of the auxiliary tree relation as black circles. Solid lines between black circles denote the auxiliary tree relation.

Input parameters of `addSetTreeRelation2` are:

- r_0: is the root of the auxiliary tree relation.
- R: contains elements of the auxiliary tree relation excluding r_0.
- P: a set of instances identifiers of added sub-trees.
- $r2P$: a map from an elements of the auxiliary tree relation to a set of instances identifiers of added sub-trees to the element.
- $p2r$: an inversion of $r2P$.
- $r2R$: a map from an elements of the auxiliary tree relation to a set of its children in the auxiliary tree relation.
- $c2r$: an inversion of $r.f_{\texttt{Desc}}$ for each r from R_0. We use it as a witness function to define an auxiliary map $r2C$.

`removeSetTreeRelation2` is dual to `addSetTreeRelation2`. It removes a set of sub-trees which possibly have different parents. The only limitation is that we cannot remove a sub-tree from another removed sub-tree. We can think about the operation as a generalization of
`removeSetTreeRelation1`.

Most of the input parameters of `removeSetTreeRelation2` have the same meaning as for
`addSetTreeRelation2`. The only extra parameter is f_{PEL}. f_{PEL} for each element of a removed sub-tree returns the root of the sub-tree. We use f_{PEL} as a witness function to compute the union of elements of removed sub-trees. Another difference is that P are elements of the same input tree instance. $r2P$ and $p2r$ are not among input parameters because information about them can be extracted from the `Child` and `parent` relations.

The third level is defined in **Appendix B.3**. It consists of
`addSetTreeRelation3` which is defined in **Appendix B.3.1** and
`removeSetTreeRelation3` which is defined in **Appendix B.3.2**.

`addSetTreeRelation3` generalizes `addSetTreeRelation2` to the addition of an arbitrary set of relations which merge input trees into one output tree. As auxiliary input information the operation requires a description of a tree-of-trees relations. Elements of the relation are merged trees. Two trees are in the relation if and only if one of them is added as a sub-tree to the other one. On Figure 4.27 we depict a result of the operation application in the case when the height of the tree-of-trees is two. The triangles of the different sizes denote trees from the different levels of the tree-of-trees relation. The biggest triangle denotes the root of the tree-of-trees relation. Trees from the second level of the tree-of-trees relation, which are denoted by the triangles of the medium size, are added as sub-trees to various elements of the root of the tree-of-trees relation. The smallest triangles denote trees which belong to the third level of the tree-of-trees relation. Additionally to the tree-of-trees relation a client has to specify auxiliary tree relations for each tree. Similarly to the `addSetTreeRelation2` we denote nodes of auxiliaries tree relations as black circles and relations as solid lines.

Input parameters of `addSetTreeRelation3` are

- r_0: a map form a tree identifier to the root of the auxiliary tree relation of the tree.
- R_{All}: contains elements of the auxiliary tree relations of all trees.
- P: a map form a tree identifier to set of trees added to the tree. From the other perspective the relation map an element of the tree-of-trees relation into its child in the relation.
- $r2P$, $p2r$, $r2R$, and $c2r$: have the same meaning as in `addSetTreeRelation2` operation.

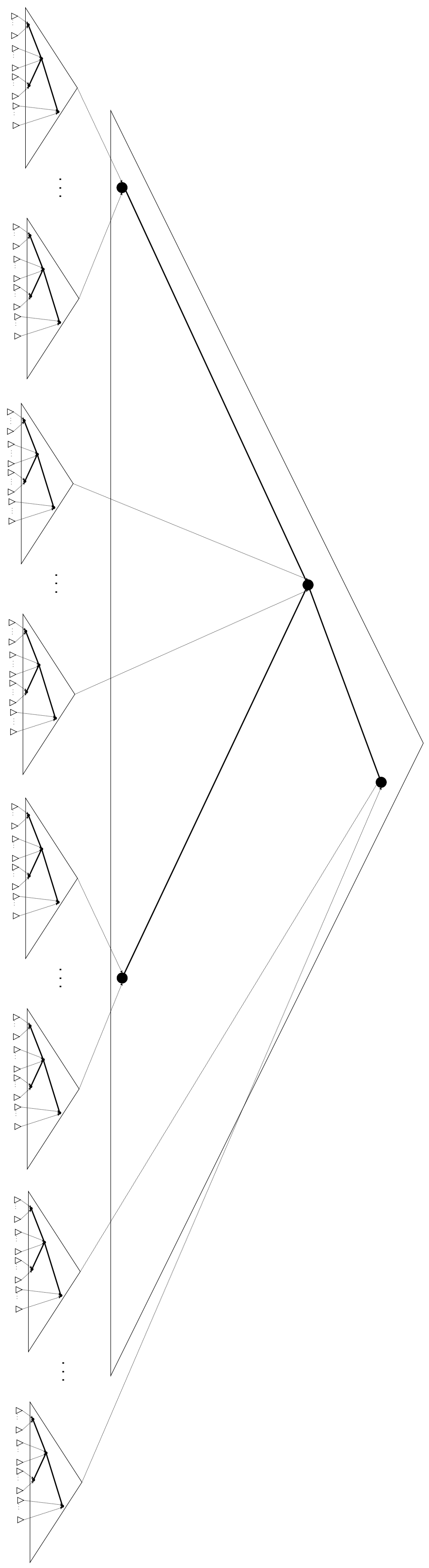


Figure 4.27: The figure depicts state of the stereotype slice after an execution of the `addSetTreeRelation3` operation. Here we denote trees as triangles. Dotted directed lines denote freshly added relations. Black circles depict elements of the tree to which we add new relations. Solid lines between black circles denote auxiliary tree relation.

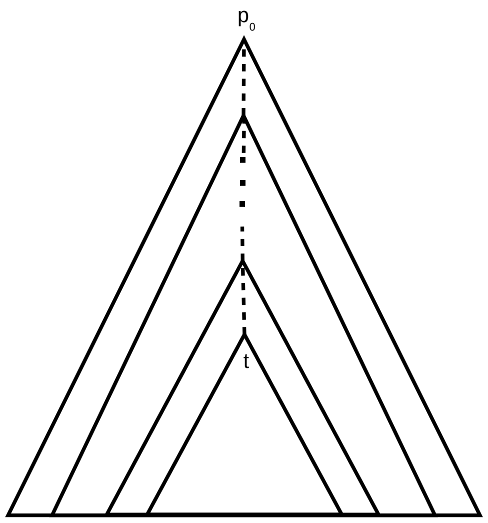
- T: set of instances identifiers of tress which participate in the operation execution.
- t_0: the root of the tree-of-trees relation.
- tD: a map of an element of the tree-of-trees relation into its descendant in the relation.
- f_{tD}: is an analog of $f_{\texttt{Desc}}$ for the tree-of-trees relation. For each pair $\langle t, t' \rangle$ of elements of the tree-of-trees relation, where t' is a descendant but not a child of t in the tree-of-trees relation, $f_{tD}[t, t']$ returns an element t'' of the relation such that t'' is a child of t and an ancestor of t' in the relation. We use f_{tD} as a witness function in some definitions.

`removeSetTreeRelation3` is dual to
`addSetTreeRelation3`. It removes an arbitrary set of relations. We can think about the operation as a generalization of
`removeSetTreeRelation2`. Similarly to `addSetTreeRelation3` a
client of the operation has to define the tree-of-trees relation and for each removed sub-tree an auxiliary tree relation.

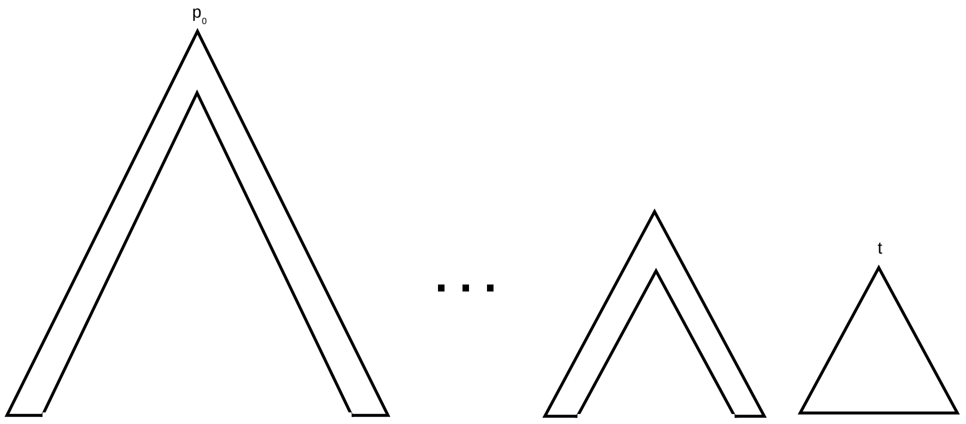
Input parameters of `removeSetTreeRelation3` are

- p_0: the root of the tree-of-trees relation.
- $p2r_0$: a map form the root of an output tree to the root of the auxiliary tree relation of the output tree.
- R: contains elements of the auxiliary tree relations of all removed trees.
- P: a set of roots of output trees.
- $r2R$, $c2r$: have the same meaning as in `removeSetTreeRelation2` operation.
- f_{PEL}: for each root of an output tree $f_{PEL}[p, .]$ returns f_{PEL} for corresponding tree.
- $p2P$: a map form a root of an output tree to a set of roots of trees removed from the tree. From the other perspective the map is a child of relation induced by the tree-of-trees relation.

As an example of the tree universal transformation let us consider a removal of a path from a tree. The operation looks artificial; nevertheless it is crucial for PIP and union-find set data structure. We consider these examples in the next section. The operation is defined on Figure 4.29. The state of the stereotype slice before the operation execution is depicted on Figure 4.28(a). The only input parameter t of `removePathTree` is the initial



(a) State of the stereotype slice before the operation execution.



(b) State of the stereotype slice after the operation execution.

Figure 4.28: The figure depicts an execution of the operation `removePathTree`. Here we denote trees and sub-trees as triangles. Dotted lines denote the removed path.

element of the removed path. The path goes from t to the root of the tree. The operation execution results in the splitting of the input tree into a set of the trees where each output tree corresponds to an element of the path. The state of the stereotype slice after the operation execution is depicted on Figure 4.28(b). The operation is specialization of `removeSetTreeRelation3`. The input parameters of `removeSetTreeRelation3` defined in the following way:

- p_0: the root of the tree-of-trees relation is the root of the input tree.

```
removePathTree⟨Tree⟩($t$ : $ref!$)
Local variables:
    $p_0 = t.\texttt{instID}$ ;
    $\forall p : p2r_0[p] = p$ ;
    $R = t.\texttt{Anc}$ ;
    $P = t.\texttt{Anc} \cup \{p\}$ ;
    $\forall r : r2R[r] = \varnothing$ ;
    $\forall c : c2r[c] = \mathbf{null}$ ;
    $\forall p, p' : f_{PEL}[p][p'] = p.f_{\texttt{Desc}}[p']$ ;
    $\forall p : p2P[p] = p.\texttt{Child} \cap t.\texttt{Anc}$ ;
{
    removeSetTreeRelation3($p_0, p2r_0, R, P, r2P, c2r, f_{PEL}, p2P$) ;
}
```

Figure 4.29: `removePathTree` stereotype operation.

- $p2r_0$: for each output tree the root of the auxiliary tree relation is equal to the root of the output trees.
- R: a set of element of all auxiliary tree relations are equal to elements of the removed path excluding the first element of the path.
- P: elements of the tree-of-trees is equal to elements of the removed path.
- $r2R$: for each output tree there is only one element in the auxiliary tree relation of the output tree; the root of the relation. Therefore, the set of children is empty for each element of each auxiliary tree relations.
- $c2r$: since set of children is empty for each element of each auxiliary tree relation, $c2r$ is never used, which implies that the value of $c2r$ does not matter. We define it as equal to **null** for all input values, but could provide any other definition.
- f_{PEL}: since for each element of the tree-of-trees relation the parent of the element in the relation is also the parent of the element in the tree f_{PEL} is equal to $f_{\texttt{Desc}}$.
- $p2P$: we define the set of children of an element of the tree-of-trees as a child of the element which belongs to the removed path.

Chapter 5

Examples of stereotype-based verification

In this chapter we consider several examples of stereotype-based verification:

- In **Section 5.1** we provide the specification of a doubly-linked list with the iterator.
- In **Section 5.2** we provide the specification of the composite design pattern.
- In **Section 5.3** we provide the specification of the data structure disjoint-set forests [33].
- In **Section 5.4** we provide the specification of the priority inheritance protocol which we have introduced in **Chapter 2**.

We conclude the chapter by considering how stereotype-based verification can be implemented in **Boogie**. In **Subsection 5.5** we provide the description of the verification of several examples which are specified using stereotypes.

5.1 List

The first example which we consider is a list with iterator. The example combines a well known data-structure, the singly-liked list [33], and the iterator design pattern [50]. The following classes participate in the example:

- **ListNode** is a class whose instances are the elements of the list.
- **List** is the wrapper class of the list. It encapsulates the list nodes from the other objects and provides an external interface to the list operations.

```
class List participate ⟨RelationInversion : NodesSlice,
RelationInversion : IteratorsSlice⟩{
    ListNode firstElement;

    List()
    stereotype createSink⟨NodesSlice⟩(this);
    stereotype createSink⟨IteratorsSlice⟩(this);
    frame ∅;
    {
        createSink⟨NodesSlice⟩(this);
        createSink⟨IteratorsSlice⟩(this);
        firstElement = null;
    }

    Iterator getIterator()
    frame ∅;
    stereotype addFreshSourceIR⟨IteratorsSlice⟩
        (result, getSink(currentElement));
    {
        return new Iterator(firstElement);
    }
}

class ListNode participate ⟨Sequence, RelationInversion : NodesSlice⟩
{
    int value;
    ListNode next;
}

class Iterator participate ⟨RelationInversion : IteratorsSlice⟩{
    ListNode currentElement;

    Iterator(ListNode currentElement)
    stereotype addFreshSourceIR(this, getSink(currentElement));
    frame ∅;
    {
        addFreshSourceIR(this, getSink(currentElement));
        this.currentElement = currentElement;
    }

    void insertAfter(ListNode node)
    requires currentElement≠null;
    requires node≠null;
    requires node.sink = null;
    requires node.next = null;
    stereotype insertAfterSeq(this, node);
    stereotype addRelationInversion⟨NodesSlice⟩(node, this.sink);
    frame node.next, currentElement.next;
    {
        node.next = currentElement.next;
        currentElement.next = node;
        insertAfterSeq(this, node);
        addRelationInversion(node, this.sink);
    }
}
```

Figure 5.1: Source code of a list with iterator.

- **Iterator** is a wrapper class of a reference to an element of the list. It also provides some element-related list update operations.

To describe semantical and topological relations between the instances of the participating classes we use the following stereotype slices:

- `Sequence` is an unnamed stereotype slice which is used to describe the topology of the list nodes. We do not give it a name because it is used only by the instances of the class **ListNode**.

- **NodesSlice** is a stereotype slice of the type `RelationInversion` which establishes the ownership relation between a list and its nodes. We use **NodesSlice** to specify that each node element is owned by not more than one list.

- **IteratorsSlice** is a stereotype slice of the type `RelationInversion` which establishes a relation between a list and iterators which refer to a list node owned by the list.

A heap and the stereotype slices for the list with an iterator is depicted in Figure 5.2. List nodes belong to an instance of a sequence stereotype slice. An instance of **NodesSlice** contains both list nodes and the list which owns the nodes. The list field `firstElement` refers to the first list node owned by the list. On the other hand an instance of **IteratorsSlice** contains the list and all iterators which refer to the list nodes owned by the list. A field `currentElement` of an iterator contains the element of the list which is referred to by the iterator.

To establish the connections between various stereotype items and a real heap we use the following glue invariants:

- $\forall\ l : \textbf{List}, it : \textbf{Iterator} :: it.\text{currentElement} \neq \textbf{null} \Rightarrow$
  $(it \in l.\textbf{IteratorsSlice}.\mathit{Source} \Leftrightarrow (l.\text{firstElement} \neq \textbf{null} \wedge$
  $it.\text{currentElement} \in l.\text{firstElement}.\texttt{Sequence}.\mathit{Elements})$
  The first glue invariant establishes the connection between the list and the iterators. If $it.\text{currentElement} \neq \textbf{null}$ then the iterator is bound to the list by **IteratorsSlice** if and only if `currentElement` of the iterator refers to the list node owned by the list. Since the iterator can not be used to update the list node when $it.\text{currentElement} = \textbf{null}$, we do not care about the relation between the list and the iterator in this case. The invariant implies that the list can be affected only by the iterators which are tracked by the **IteratorsSlice** stereotype slice.

- $\forall\ n : \textbf{ListNode} :: n.\textbf{NodesSlice}.\mathit{sink} \neq \textbf{null} \Rightarrow$
  $n.\texttt{Sequence}.\mathit{Elements} \subseteq n.\textbf{NodesSlice}.\mathit{sink}.\textbf{NodesSlice}.\mathit{Source}$
  The second glue invariant establishes an ownership relation between the list and the list nodes. If a node is owned by the list then the

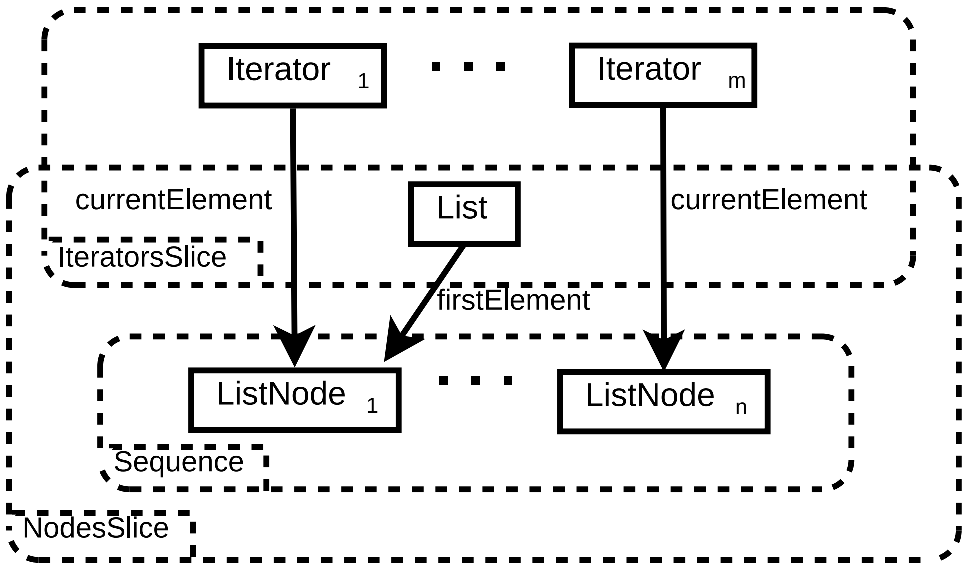


Figure 5.2: The figure depicts the heap and the stereotype slices for the list with the iterator example. Class instances are denoted by rectangles with class names inside the rectangles. The heap references are denoted by arrows. Each arrow is labeled by the corresponding field name. Dotted rounded rectangles enclose elements of the stereotype instances. At the left bottom corner of each dotted rounded rectangle is the name of the stereotype slice to which the stereotype instance belongs.

same is true for all other nodes from the same sequence stereotype instance. The invariant is equivalent to the following one: all nodes from the same sequence stereotype instance have the same list owner. The invariant implies that list nodes which are owned by different lists are disjoint.

- $\forall\ n$: **ListNode** :: n.`next` $=$ n.`Sequence.next`
  The third glue invariant establishes the relation between **ListNode** fields and the sequence stereotype. Essentially it states that the `next` field of **ListNode** is always equal to the stereotype item `next` of the `Sequence` stereotype. The invariant implies that the list nodes form a linked list in the heap.

Before we move to the list and the iterator specification let us consider auxiliary stereotype operations which we use in the specifications. We define the following auxiliary operations in Figure 5.3:

- `addFreshSourceIR` updates a `RelationInversion` stereotype slice. First it creates a fresh singleton stereotype slice for o' and then if

```
addFreshSourceIR⟨RelationInversion⟩($o$ : $ref$, $o'$ : $ref$)
{
    createSource(o');
    if(o≠ null) then addRelationInversion(o', o);
}

insertAfterSeq⟨Sequence⟩($o$ : $ref$, $o'$ : $ref$)
Local variables:
    nextEl = old(o.next);
{
    if(nextEl ≠ null) then removeSequenceRelation(o);
    addSequenceRelation(o, o');
    if(nextEl ≠ null)
    then addSequenceRelation(o', nextEl);
}

getSink: $ref \rightarrow ref$
getSink(o) = if(o≠ null) then o.sink else null;
```

Figure 5.3: Auxiliary stereotype operations and functions which are used to specify the list with the iterator.

o is not null establishes the relation between o' and o. After the operation execution if o is not null then o' is a source and o is a sink of the same stereotype instance, otherwise o' is an element of a singleton stereotype instance.

- `insertAfterSeq` inserts o' immediately after o in the sequence stereotype. The operation checks whether o is the right tail of the stereotype instance in which it participates, which is the case when `nextEl` = **null**. In this case we use `addSequenceRelation` to add the relation between o and o'. Otherwise we split a sequence in which o participates in two sub-sequences. The first sequence contains all elements from the beginning of the initial sequence until o, and the seconde one contains all elements from `nextEl` until the end of the initial sequence. After the sequence split we add the relation between o and o'. Then we add a relation between the end of the sequence which contains o' and `nextEl`. The operation does not require o' to be a singleton sequence. If o' belongs to a non-singleton sequence then the whole sequence is inserted between o and o.`next`.

- `getSink` is an auxiliary function which returns a sink of the input object if the object is not null and returns null otherwise.

Now we can look at the source code and specifications of the list with iterator which are presented in Figure 5.1.

The header of the class **List** declares that the class participates in the **NodesSlice** and **IteratorsSlice** stereotype slices. The class has a field `firstElement` which refers to the first list node.

The class constructor sets `firstElement` to **null** and calls stereotype constructors to initiate stereotype items of **NodesSlice** and **IteratorsSlice**. After the constructor execution the sources of both **NodesSlice** and **IteratorsSlice** are empty, which means that the list does not own any list nodes and there is no list iterator which refers to the list nodes. The same stereotype constructors are used in the constructor specification. Since the constructor affects only the fields of the freshly allocated object, the frame is empty.

The `getIterator` method creates an iterator which refers to the first list node. In the method specification we use `addFreshSourceIR` to specify the method behavior. The specification duplicates the corresponding specification of the iterator constructor.

The **ListNode** class participates in the unnamed `Sequence` stereotype and in **NodesSlice**. The class contains two fields: `next` refers to the next list node, and `value` contains an integer value of the list node. The class does not have any methods or constructors.

Class **Iterator** participates in the **IteratorsSlice** stereotype slice. The only field of the class `currentElement` contains a list node which is referred to by the iterator.

The constructor sets the value of `currentElement` and calls the `addFreshSourceIR` stereotype operation to register the freshly allocated iterator in **IteratorsSlice**. It is possible that `currentElement` is null. In this case `getSink` returns null and the iterator is not bound to any list.

The `insertAfter` method inserts an input node into the list immediately after the list node which is referred to by the iterator. The method precondition guarantees that the input parameter `node` belongs to a singleton sequence instance which is not owned by a specific list. The pre-conditions guarantee as well that the iterator refers to a list node. The first two lines of the method body change the heap in such a way that `node` is inserted in the list. Then the `insertAfterSeq` call updates the `Sequence` stereotype in a consistent way. And finally `addRelationInversion` is used to set the owner of `node` to the list. `insertAfterSeq` and `addRelationInversion` are duplicated in the method specification.

5.2 Composite

In this sub-section we consider the composite design pattern [50]. The pattern is used to represent a modeled entity as a composition of sub-entities.

```
class Container{
  Elements: $Reg$;

  Container()
  ensures $Elements = \varnothing$;
  frame $\varnothing$;


  void addEl(o: Composite)
  ensures $Elements = \texttt{old}(Elements) \cup \{o\}$;
  frame $this.Elements$;

  void removeEl(o: Composite)
  ensures $Elements = \texttt{old}(Elements) \setminus \{o\}$;
  frame $this.Elements$;
}
```

Figure 5.4: Specification of the **Container** class.

```
Bool invCompositeUnpacked(o: Composite, delta: $\mathbb{Z}$)
invCompositeUnpacked($o, delta$) $\Leftrightarrow o.numberOfDescendants =$
      $(\sum_{o' \in o.\texttt{Child}} o'.numberOfDescendants) + |o.\texttt{Child}| + delta$

Bool invComposite(o: Composite)
invComposite($o$) $\Leftrightarrow$ invCompositeUnpacked($o, 0$)
```

Figure 5.5: **Composite** behavioral invariant and invariant state.

Each of the sub-entities is also composed from entities. The entity composition forms a tree relation. An entity can contain aggregated information about its descendants, e.g., the number of descendants. From a behavioral perspective the most interesting aspect is the update of this aggregated information. Such an update can have a global effect: for instance when we add a new child to a component we have to update the number of descendants for all ancestors of the component.

The implementation of the composite pattern has to track the set of children of a composite. In order to do this we need a container which contains a set of objects. We provide a possible interface for a container in Figure 5.4. The interface of the container contains the ghost field `Elements` of the region type. The field contains all elements of the container. The constructor creates an empty container. The method `addEl` adds an element to the container. The method `removeEl` removes an element from the

```
class Composite participate ⟨Tree⟩{
  owner: Composite;
  Children: Container;
  numberOfDescendants: int;

  Composite()
  ensures invComposite(this);
  stereotype createSingletonTree(this);
  frame ∅;
  {
    createSingletonTree(this);
    owner = null;
    Children = new Container();
    numberOfDescendants = 0;
  }

  void addComponent(child : Composite)
  requires child≠null;
  requires this∉child.Elements;
  requires child.root = child;
  requires ∀ o ∈ this.Anc ∪ {this} : invComposite(o);
  ensures ∀ o ∈ this.Anc ∪ {this} : invComposite(o);
  frame (this.Anc ∪ {this}×numberOfDescendants)∪{Children, child.owner};
  stereotype addTreeRelation(child, this);
  {
    child.owner = this;
    Children.addEl(child);
    addTreeRelation(child, this);
    compositeUpdate(−child.numberOfDescendants −1);
  }

  void removeComponent(child : Composite)
  requires child∈this.Child;
  requires ∀ o ∈ this.Anc ∪ {this} : invComposite(o);
  ensures ∀ o ∈ this.Anc ∪ {this} : invComposite(o);
  frame (this.Anc ∪ {this}×numberOfDescendants)∪{Children, child.owner};
  stereotype removeTreeRelation(child);
  {
    child.owner = null;
    Children.removeEl(child);
    removeTreeRelation(child, this);
    compositeUpdate(child.numberOfDescendants + 1);
  }

  void compositeUpdate(delta : int)
  requires invCompositeUnpacked(this, delta);
  requires ∀ o ∈ this.Anc : invComposite(o);
  ensures ∀ o ∈ this.Anc ∪ {this} : invComposite(o);
  frame Anc ∪ {this}×numberOfDescendants;
  {
    numberOfDescendants −= delta;
    if (owner ≠ null)
        owner.compositeUpdate(delta);
  }
}
```

Figure 5.6: Specification of the **Composite** class.

container. Both of them update the `Elements` field in the obvious way. The container can be implemented in various ways, for instance, as a linked list or a dynamic array. Since it is not relevant to the composite pattern we leave it unspecified.

The composite pattern is implemented as **Composite** class (see Figure 5.6). The class participates in the unnamed tree stereotype slice. The slice is used to formalize the component containment relation as a tree. The class has the following fields:

- `owner` refers to the parent of the object in the composite tree.
- `Children` contains the set of children of the object in the composite tree.
- `numberOfDescendants` contains the total number of descendants of the object in the composite tree. An object is a descendant of another object in the composite tree if and only if the second object is reachable from the first object via the transitive closure of `owner`.

Since the entire topology is described by only one stereotype, the glue invariant is quite simple:

- $\forall\ o$: **Composite** :: $o.owner = o.$`parent`. Field *owner* is always equal to `parent` of the tree stereotype slice.
- $\forall\ o$: **Composite** :: $o.Children.Elements = o.$`Child`. The content of field *Children* is always equal to `Child` of the tree stereotype slice.

Essentially the glue invariants state that the composite tree is the same as the tree defined by the corresponding stereotype instance.

The last thing which we need to specify is the behavioral invariants and the invariant states of the composite design pattern. The behavioral invariants specify the properties of the fields of non-reference types. For the composite pattern these are the properties of the field `numberOfDescendants`. The behavioral invariants are straightforward (see Figure 5.5):

- the behaviorial state of the object o is $delta \in \mathbb{Z}$ if and only if the value of `numberOfDescendants` differs from the actual number of descendants in exactly *delta*.
- the behaviorial invariant holds for a composite o (denoted as `invComposite`(o)) if and only if the behaviorial state of the object o is 0 (denoted as `invCompositeUnpacked`($o, 0$))

We use the invariant state to simplify the usage of the class specification. Instead of the the exposing internal representation (`numberOfDescendants` field) and duplicating parts of the invariant specifications we use invariant states.

Now we can look at the source code and the specifications of the composite which are presented in Figure 5.6.

The header of the class **Composite** declares that the class participates in the unnamed `Tree` stereotype slice. As we have mentioned above the class has the following fields: `owner`, `Children`, and `numberOfDescendants`.

The class constructor sets `owner` to **null**, allocates a new empty container referred to by `Children`, and sets numberOfDescendants into 0. The stereotype constructor is used to update the values of the stereotype items. The constructor execution results in a singleton composite allocation. The constructors specification uses the stereotype constructor. Since the values of the class fields can be inferred from the stereotype item values and from the invariants, the constructors specification does not mention them.

The addComponent method adds a sub-component to a composite. The only input parameter `child` refers to the added sub-component. The pre-condition guarantees that `child` is the root of a tree instance which is disjoint from the tree to which the method receiver belongs. The method also requires and preserves the behavioral invariant for the method's receiver and its ancestors. The first line of the method body updates the heap, the second one updates the set of children, and the third one updates the stereotype slice. These updates keep the heap and the stereotype slice consistent. The last line calls the auxiliary method *compositeUpdate* which updates the values of `numberOfDescendants` for the method's receiver and its ancestors. The update restores the behavioral invariants of the method's receiver and its ancestors which were broken by the addition of the sub-component. The only parameter of the *compositeUpdate* method call is the number of descendants which were removed from the method's receiver. The method specification uses the stereotype operation `addTreeRelation` to specify the method effects.

The method removeComponent is symmetric to the method addComponent. Instead of adding the sub-component it removes it. The pre-condition checks that the input parameter `child` is a child of the method's receiver. The rest of the method is similar to addComponent; that is why we omit its description.

The last method is *compositeUpdate* which restores the behavioral invariants which were broken by addComponent or by removeComponent. It takes a single input parameter *delta* and assumes that the behavioral invariant state of the method's receiver and its ancestors are *delta*. The post-condition of the method states that the behavioral invariants of the method's receiver and its ancestors were restored. The first line of the method body restores the behavioral invariant of the method's receiver. If the method's receiver is not the root of the tree then the recursive call of the method *compositeUpdate* restores the behavioral invariants of the receiver's ancestors.

```
compressTreePath⟨Tree⟩(o : ref)
Local variables:
    l = old(o.root);
    P = old(o.Anc ∪ {o}) \ {l};
{
    if (o.root≠o)
    then {
        removePathTree(o);
        addSetTreeRelation1(l, P);
    }
}
```

Figure 5.7: compressTreePath stereotype operation.

```
unionTree⟨Tree⟩(o : ref, o′ : ref)
{
    compressTreePath(o);
    compressTreePath(o′);
    if (o ∉ o′.Elements) then addTreeRelation(o.root, o′.root);
}
```

Figure 5.8: unionTree stereotype operation.

5.3 Disjoint-set forests

In this section we consider the disjoint-set forests [33] data structure, **DSF** in short. This data structure is used as an efficient representation of a set of disjoint sets. **DSF** provides two operations: the union of two sets and finding to which set a provided element belongs. Both operations have amortized logarithmic complexity. This efficiency is achieved by using lazy computations.

Let us consider how the **DSF** is organized. Each set is represented by a disjoint tree. The tree structure is implemented by a backward reference from a child to a parent. For implementation purposes the root of the tree points to itself. The root of a tree is used as a set identifier. To extract the identifier of a set to which a specific element belongs it is enough to traverse the tree from the element to the root. A union of two disjoint sets is done in a lazy way: a backward reference is added from the set identifier of to the set identifier of another one. In this way the two trees which represent the sets are merged into one. To achieve the desired amortized complexity a tree path compression is done during the extraction of the set identifier. By tree path compression we mean that all elements of a tree path are removed

```
class SetElement participate ⟨Tree⟩{
    owner: SetElement;

    SetElement()
    stereotype createSingletonTree(this);
    frame ∅;
    {
        createSingletonTree(this);
        owner = this;
    }

    SetElement find()
    stereotype compressTreePath(this);
    frame this.Anc ∪ {this}×owner;
    {
        if (owner = this) {
            return this;
        } else {
            SetElement r = owner.find();
            owner = r;
            removeTreeRelation(this);
            addTreeRelation(this, r);
            return r;
        }
    }

    union(s: SetElement)
    stereotype unionTree(this, s);
    frame (this.Anc∪s.Anc ∪ {this,s}×owner)∪{owner};
    {
        SetElement tr, sr;
        tr = find();
        sr = s.find();

        if (tr != sr) {
            owner = s;
            addTreeRelation(this, s);
        }
    }

}
```

Figure 5.9: An implementation and specification of disjoint-set forests.

from the tree and added as children to the root.

To implement **DSF** we use class SetElement which represents an element of a disjoint set. The only field of the class, `owner`, points to the parent of the element in the tree. To specify the tree relation we use the unnamed tree stereotype slice.

The glue invariant establishes the relation between the tree formed by the `owner` field and the tree stereotype slice.

- $\forall\ el : \text{SetElement} :: o \neq o.\texttt{root} \Rightarrow o.\text{owner} = o.\texttt{parent}$. For each set element excluding the root, `owner` refers to the parent of the element.

- $\forall\ el : \text{SetElement} :: o = o.\texttt{root} \Rightarrow o.\text{owner} = o$. For the root `owner` is a self reference.

To specify **DSF** we introduce two auxiliary tree stereotype operations. The first one is `compressTreePath`. It is defined in Figure 5.7. The operation specifies how a tree stereotype changes after a path compression. The only input parameter refers to the beginning of the compressed path. If the input parameter is the root of a tree then nothing has to be done. Otherwise at first we remove the path from the tree using the stereotype operation `removeTreeRelation` and then we use `addSetTreeRelation1` stereotype operation to add all removed elements to children of the root. Both `removeTreeRelation` and `addSetTreeRelation1` are defined and discussed in details in **Appendix B**.

The second auxiliary operation is `unionTree`. It is defined in Figure 5.8. It describes the aggregated update of the tree slice by the union of sets to which the input parameters o and o' belong. At first the operation calls `compressTreePath` to compress the tree paths which start at o and o' respectively. If o and o' belong to different sets then the sets are merged by adding a relation between the roots of the trees which they represent.

The implementation of the **DSF** is provided in Figure 5.9. The constructor creates a singleton set. The freshly allocated object is the only element of the set and the root of the representation tree. The tree stereotype constructor is used for both stereotype items initialization and method specification.

The `find` operation returns the set identifier of the set to which the method receiver belongs. If the receiver is the root of the tree then it is also the set identifier. In this case the method returns the receiver and terminates. Otherwise the recursive call is done. To obtain the set identifier and compress the tree path starting from the parent of the method receiver. The next operation performs the path compression for the method receiver. Essentially it moves the receiver to the children of the root. The next two stereotype operations update the tree stereotype slice to make it consistent with the updated heap and restore the glue invariant. The last statement returns the set identifier.

`union` merges the set to which the method receiver belongs with the set to which the input parameter belongs. The fist two statements get the identifiers of the merged sets. If the sets are disjoint, which is checked by comparison of their identifiers, then the sets are merged. To merge the sets it is enough to set the parent of the receiver to the set identifier of the input parameter. `addTreeRelation` is used to restore the glue invariant. Otherwise if the receiver and the input parameters belong to the same set, nothing has to be done.

5.4 PIP

The last example which we consider is the priority inheritance protocol which we introduced in **Chapter 2**.

Let us begin with the **PIP** topology. The heap topology of **PIP** is formed by the *blockedBy* field. To specify it we use the unnamed tree and the unnamed sequence stereotypes. An example of a heap and stereotypes slice layout of **PIP** is provided in Figure 5.10. If there are no deadlocks then *blockedBy* forms a tree relation. By acquiring and releasing nodes we add and remove sub-trees. We specify this aspect of the **PIP** with the help of the unnamed tree stereotype. From the tree slice perspective **PIP** nodes form a forest. On the other hand if a node o attempts to acquire a node o' which is already locked by one of o's descendants then a deadlock is created. To specify this behavior we remove the path between o and o' from the tree instance and add it to a cyclic list. To specify this cyclic list we use the sequence stereotype slice. From the sequence preceptive if nodes form a deadlock then they belong to the same non-singleton sequence instance. A node which does not participate in a deadlock forms a singleton sequence.

To distinguish these two possible states we introduce the **PIP** glue invariant state `st`. `st` is an enumeration type whose values are $\texttt{st}_L$ and $\texttt{st}_D$. A **PIP** node is in a deadlock if and only if its glue invariant state is equal to $\texttt{st}_D$. Otherwise its glue invariant state is equal to $\texttt{st}_L$. We define the semantics of $\texttt{st}_L$ and $\texttt{st}_D$ by means of glue invariants below.

To specify **PIP** we need the transitive closure and the inversion of the *blockedBy* field. Since we would like to avoid the explicit transitive closure usage we use stereotype items to define the functions which simulate this construct. The definitions are provide in Figure 5.11.

- $\texttt{blockedBy}^{-1}$ is the inversion of the *blockedBy* field. If a node is not in a deadlock state then $\texttt{blockedBy}^{-1}$ is equal to the node's children in the three stereotype. Otherwise we add the previous element of the node in the sequence stereotype to the result.

- $\texttt{blockedBy}^{*}$ is the transitive and reflexive closure of the *blockedBy* field. If a node is not in a deadlock state then $\texttt{blockedBy}^{*}$ is equal to

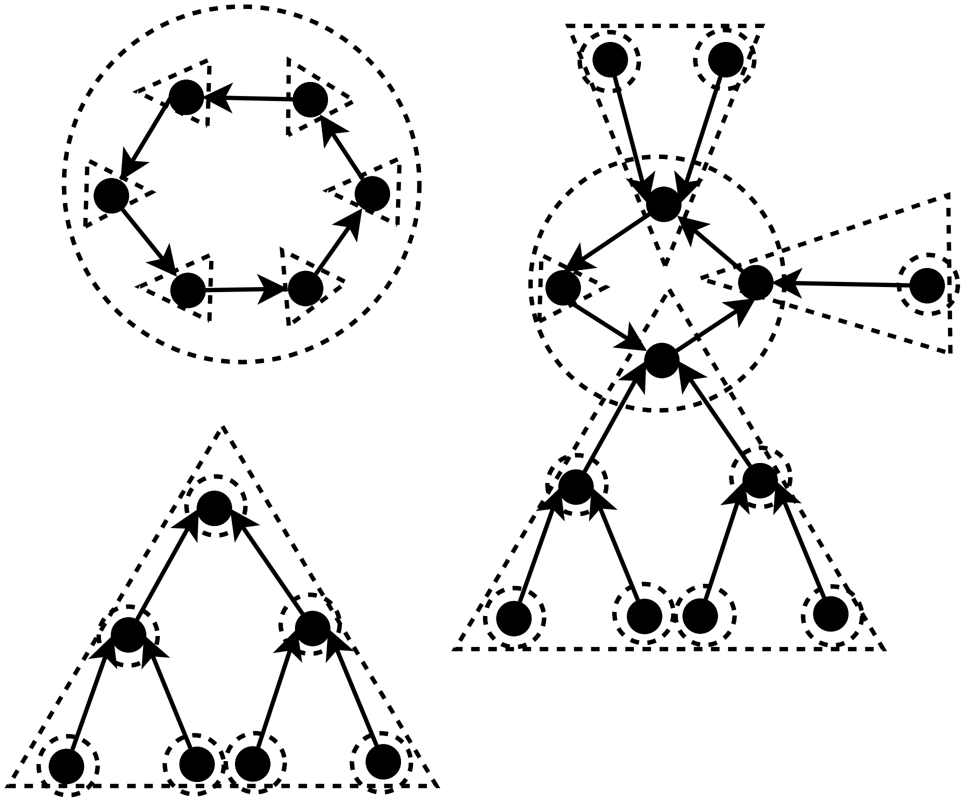


Figure 5.10: The figure depicts a heap and stereotype slices layout for the PIP example. PIP nodes are denoted as black circles. Black arrows denote the blockedBy field. Dotted triangles enclose elements of the tree stereotype instances. Dotted circles enclose elements of the sequence stereotype instances.

the ancestors of the node in the tree stereotype plus the node itself. If a node is in a deadlock state then we add to the result all nodes which participate in the deadlock. The set of nodes which participate in the deadlock is accessible via elements of the sequence stereotype instance of the root of the tree instance to which the node belongs. If there is no deadlock the root of the tree forms a singleton sequence whose elements contain only the root. Therefore we do not need to explicitly consider two cases and always can add the elements of the sequence stereotype instance of the root to the result.

- `blockedBy`$^{+}$ is the transitive and irreflexive closure of the *blockedBy* field. We construct it from the reflexive transitive closure by removing the node itself from the result.

Let us define the **PIP** invariants:

$$\begin{array}{l}
\textit{Reg}\ \texttt{blockedBy}^{-1}(\mathrm{o}:\ \mathrm{PIPNode}) \\
\texttt{blockedBy}^{-1}(o)\ =\ \texttt{if}(o.\texttt{st} = \texttt{st}_L)\ \texttt{then}\ o.\texttt{Tree.Child} \\
\texttt{else}\ o.\texttt{Tree.Child} \cup \{\texttt{cyclicPrev}(o)\} \\
\\
\textit{Reg}\ \texttt{blockedBy}^{*}(\mathrm{o}:\ \mathrm{PIPNode}) \\
\texttt{blockedBy}^{*}(o)\ =\ o.\texttt{Tree.Anc} \cup o.\texttt{Tree.root.Sequence.}\textit{Elements} \cup \{o\} \\
\\
\textit{Reg}\ \texttt{blockedBy}^{+}(\mathrm{o}:\ \mathrm{PIPNode}) \\
\texttt{blockedBy}^{+}(o)\ =\ \texttt{blockedBy}^{*}(o) \setminus \{o\}
\end{array}$$

Figure 5.11: `blockedBy` predicates.

- $\forall\ o : \text{PIPNode} :: o.\texttt{st} = \texttt{st}_L \Leftrightarrow o.\textit{blockedBy} = o.\texttt{Tree.parent}$. This invariant defines the meaning of the locked state. An object is in the locked state if and only if its field *blockedBy* plays the role of the parent in the tree stereotype slice.

- $\forall\ o : \text{PIPNode} :: o.\texttt{st} = \texttt{st}_D \Leftrightarrow o.\textit{blockedBy} = \texttt{cyclicNext}(o)$. This invariant defines the meaning of the deadlocked state. An object is in the deadlocked state if and only if its field *blockedBy* plays the role of the next element in the tree stereotype slice.

- $\forall\ o : \text{PIPNode} :: o.\texttt{st} = \texttt{st}_L \Rightarrow \texttt{isSingletonSequence}(o)$. If an object's state is locked then the instance of the sequence in which it participates is a singleton.

- $\forall\ o, o' : \text{PIPNode} :: o' \in o.\texttt{Tree.Desc} \Rightarrow o'.\texttt{st} = \texttt{st}_L$. If an object is a descendant of some other object in a tree stereotype then the object does not participate in a deadlock.

- $\forall\ o, o' : \text{PIPNode} :: o.\texttt{st} = \texttt{st}_D \wedge o' \in o.\texttt{Sequence.}\textit{Elements} \Rightarrow o'.\texttt{st} = \texttt{st}_D$. If two objects participate in the same sequence instance and one of them participates in a deadlock then the other one also participates in a deadlock.

On top of the topology invariants we can define the behavioral invariant. The definitions are provided in Figure 5.12:

- `invPriorities` defines the semantics of the `priorities` fields. If the **PIP** node invariant holds then `priorities` is a multiset which contains the current priorities of all **PIP** nodes which are blocked by this one. Nevertheless it is possible that during the change of the **PIP** topology the value of `priorities` is different form the intended one. Input parameters $from$ and *to* specify this difference. `invPriorities` guarantees the following:

```
Bool invPriorities(o: PIPNode, from: $\mathbb{N}$, to: $\mathbb{N}$)
invPriorities$(o, from, to) \Leftrightarrow$
    $(\forall i \in \mathbb{N}^+ \setminus \{from, to\} : priorities[i] \ =$
        $|\{o' \in \texttt{blockedBy}^{-1}(o) \ : \ o.currentPriority = i\}|) \wedge$
    $(from \neq 0 \Rightarrow priorities[from] \ =$
        $|\{o' \in \texttt{blockedBy}^{-1}(o) \ : \ o.currentPriority = from\}| + 1) \wedge$
    $(to \neq 0 \Rightarrow priorities[to] \ =$
        $|\{o' \in \texttt{blockedBy}^{-1}(o) \ : \ o.currentPriority = to\}| - 1)$

Bool invCurrentPriority(o: PIPNode)
invCurrentPriority$(o) \Leftrightarrow$
    $o.currentPriority = max(o.priorities \cup \{o.defaultPriority\})$

Bool invDefaultPriority(o: PIPNode)
invCurrentPriority$(o) \Leftrightarrow defaultPriority > 0$

Bool invPIPunpacked(o: PIPNode, from: $\mathbb{N}$, to: $\mathbb{N}$)
invPIPunpacked$(o, from, to) \Leftrightarrow$
    invPriorities$(o, from, to) \wedge$ invDefaultPriority$(o)$

Bool invPIP(o: : PIPNode)
invPIP$(o) \Leftrightarrow$ invPIPunpacked$(o, 0, 0) \wedge$ invCurrentPriority$(o)$
```

Figure 5.12: PIPNode invariants

  - For all priorities i which are distinct from $from$ and to `priorities`$[i]$ is equal to the number of nodes whose are blocked by this one and which current priority is equal to i.
  - if $from$ is not equal to 0 then the value of `priorities`$[from]$ is smaller by one than it is supposed to be.
  - if to is not equal to 0 then the value of `priorities`$[to]$ is greater by one than it it is supposed to be.

- `invCurrentPriority` states that the value of the current priority is equal to the maximum of priorities which belong to `priorities` and the default priority of the node.
- `invDefaultPriority` states that the default priority of a **PIP** node is strictly positive.
- `invPIPunpacked` is a composition of `invPriorities` and `invCurrentPriority`. We use it to define the behavioral state of the **PIP** node. We say that the behavioral state of a node o is $\langle from, to \rangle$ if and only if `invPIPunpacked`$(o, from, to)$ holds.

- `invPIP` is the desired **PIP** node invariant. A **PIP** node invariant holds if and only the behavioral state of the node is $\langle 0, 0\rangle$ and its default priority invariant holds.

```
constructor createSingletonPIPNode⟨Tree, Sequence, st⟩ (n: ref)
{
   createSingletonTree(n);
   createSingletonSequence(n);
}
```

Figure 5.13: `createSingletonPIPNode` stereotype operation.

```
acquireBlocked⟨Tree, Sequence, st⟩ (tree: ref, subTree: ref)
Local variables:
   $Inst = tree \cup tree.\mathtt{Anc}$;
   $\forall o : nextInst[o] = o.\mathtt{parent}$;
   $\forall o : ElUnion[o] = o.\mathtt{Anc} \cup \{o\}$;
{
   if ($tree \notin subTree.\mathtt{Desc}$)
   then addTreeRelation($tree, SubTree$)
   else {
      addSetSequenceRelation($tree, subTree, Inst, nextInst, ElUnion$);
      removePathTree($tree$);
   }
}
```

Figure 5.14: `acquireBlocked` stereotype operation.

```
acquireStereotype⟨Tree, Sequence, st⟩ (tree: ref, subTree: ref)
{
   if(subTree.blockedBy = null)
   then addTreeRelation(subTree, tree)
   else acquireBlocked(tree, subTree);
}
```

Figure 5.15: `acquireStereotype` stereotype operation.

To specify **PIP** we introduce two axillary stereotype operations and one constructor:

- In Figure 5.13 we define the stereotype constructor

```
class PIPNode⟨Tree, Sequence⟩{
  PIPNode blockedBy;
  int defaultPriority;
  int currentPriority;
  MultiSet⟨int⟩ priorities;
  st {st_L, st_D};

  PIPNode(int priority)
  requires priority > 0;
  ensures invPIP(this);
  stereotype createSingletonPIPNode(this);
  {
    createSingletonPIPNode(this);
    this.st = st_L;
    blockedBy = null;
    defaultPriority = priority;
    currentPriority = priority;
    priorities = ∅;
  }
}
```

Figure 5.16: Specification of the class PIPNode and its constructor.

`createSingletonPIPNode`. The constructor calls the tree and the sequence constructors to create singleton tree and sequence instances.

- In Figure 5.14 we define the stereotype operation `acquireBlocked`. It is used to specify the stereotype updates when a blocked node $subTree$ is acquired. There are two possibilities: either the acquired node $subTree$ is blocked by a descendant of the current node $tree$ or not. In the first case we merge all nodes which belong to a tree path between $tree$ and $subTree$ into a single sequence instance. Then we remove the path between $tree$ and $subTree$ from the tree instance. And at the end we change the behavioral states of all affected **PIP** nodes to deadlocked. In the other case if the acquired node $subTree$ is blocked by a node which is not s descendant of the current node $tree$ we add a tree relation between them.

- In Figure 5.15 we define the stereotype operation `acquireStereotype`. It is used to specify the accumulated behavior of the *acquire* operation. First of all the operation checks whether the acquired node is blocked or not. In the first case the behavior is specified by the `acquireBlocked` operation. In the other case a tree relation is added between $subTree$ and $tree$.

```
void updatePriorities(from: int, to: int)
requires ∀o ∈ blockedBy+(this):invPIP(o);
requires invPIPunpacked(this, from, to);
ensures ∀o ∈ blockedBy*(this):invPIP(o);
frame blockedBy*(this)×{currentPriority, priorities};
{
  oldCurrentPriority int;
  oldCurrentPriority = currentPriority;

  if (from > 0)
    priorities = priorities\{from};
  if (to > 0)
    priorities = priorities∪{to};

  currentPriority = max(defaultPriority, max(priorities));
  if(blockedBy ≠ null && oldCurrentPriority ≠ currentPriority)
    blockedBy.updatePriorities
      (oldCurrentPriority, currentPriority);
}
```

Figure 5.17: Specification of the method updatePriorities of the class PIPNode.

Let us now specify **PIP**. In Figure 5.16 the **PIP** class and constructor definitions are provided. The class definition declares that the class participates in two unnamed stereotype slices: the tree and the sequence slices. The class definition contains the **PIP** glue invariant state `st`. The constructor uses the `createSingletonPIPNode` constructor for both method specifications and ghost updates. The only non-topological specification of the constructor is the pre-condition which checks that the priority of the input parameter is strictly positive and the post-condition which guarantees that the **PIP** invariant holds for the freshly allocated object.

The specification of *updatePriorities* is provided in Figure 5.17. The method pre-condition checks that the **PIP** invariant holds for all nodes which are reachable from the current one via $\texttt{blockedBy}^+$. The pre-condition also checks that the behavioral state of the current node is
$\langle from, to\rangle$. The post-condition states that the **PIP** invariant is restored for the current node and preserved for all nodes which are reachable via $\texttt{blockedBy}^+$. The frame property states that only *currentPriority* and *priorities* fields of the nodes which are reachable via $\texttt{blockedBy}^*$ are affected by the operation. Since the operation does not change stereotype items there is no need for stereotype operations for both method specifications and ghost updates.

```
void release(n: PIPNode)
requires this.root = this;
requires $n \in$ this.Child;
requires $\forall o \in$ blockedBy$^*$(n):invPIP(o);
ensures $\forall o \in$ blockedBy$^*$(n):invPIP(o);
stereotype removeTreeRelation(n);
{
  n.blockedBy = null;
  if (n.currentPriority $\neq$ 0)
  {
    removeTreeRelation(n);
    updatePriorities(n.currentPriority, 0);
  }
}

void acquire(n: PIPNode)
requires this.root = this;
requires n $\neq$ null;
requires $\forall o \in$ blockedBy$^*$(n)$\cup$blockedBy$^*$(this):invPIP(o);
ensures $\forall o \in$ blockedBy$^*$(n)$\cup$blockedBy$^*$(this):invPIP(o);
stereotype acquireStereotype(this, n);
{
  if(n.blockedBy = null){
    addTreeRelation(n, this);
    n.blockedBy = this;
    if (n.currentPriority $\neq$ 0)
      updatePriorities(0, n.currentPriority);
  } else {
    acquireBlocked(this, n);
    if ($tree \in subTree$.Desc) $\langle Inst, \mathtt{st}, \mathtt{st}_D \rangle$;
    this.blockedBy = n;
    if (currentPriority $\neq$ 0)
      n.updatePriorities(0, this.currentPriority);
  }
}
```

Figure 5.18: Specification of the methods release and acquire of the class PIPNode.

The specifications for both *release* and *acquire* are provided in Figure 5.18. The pre-condition of *release* checks that *this* refers to a tree root in a tree stereotype slice and that the input parameter is a child of *this* in the tree stereotype slice. In other words *this* is not blocked and the input parameter refers to a **PIP** node which is owned by *this*. The method also requires and guarantees preservation of the **PIP** invariant for all nodes which are reachable via `blockedBy`*. The method uses `removeTreeRelation` for both method specification and ghost updates.

The pre-condition of *acquire* checks that *this* refers to a tree root in a tree stereotype slice and that the input parameter is not **null**. The method also requires and guarantees preservation of the **PIP** invariant for all nodes which are reachable from *this* and the input parameter via `blockedBy`*. `acquireStereotype` is used to specify the method effect on stereotype items. `acquireBlocked` and `addTreeRelation` are used for ghost updates. We can see that the method stereotype specifications duplicate the method ghost updates.

5.5 Boogie experiments

To validate our approach we directly encoded into **Boogie** [75] some of the stereotypes and examples which we have mentioned above. In our experiments we used the **Boogie** build from 2010.07.20. **Boogie** generates the proof obligations which are verified by the Z3 theorem prover. In our experiments we used version 2.10 of Z3 [39]. We discuss **Boogie** and Z3 in more details in **Chapter 7**. In this section we discuss the results of these experiments. The Boogie specification of the discussed examples can be downloaded from:
http://people.inf.ethz.ch/rudicha/Stereotypes_Boogie_Examples.zip.

Before we move on to the examples let us mention some technical problems which we have come across.

The first technical problem is related to the **SLS**-to-**SSOR** transformation which is defined in **Section 6.4**. The transformation can be easily done by an automatic tool. On the other hand, since the transformation involves an enormous amount of case splits, it is not really feasible for a manual encoding. In our experiments we have used a manual encoding into **Boogie** which prevented us from using the **SLS**-to-**SSOR** transformation. Instead we had to use the usual weakest pre-condition calculus (**WPC**) provided by **Boogie**. This replacement had several consequences:

- Since **WPC** has to deal with non-determinism and is not targeted to dealing with **SLS**, we believe that it is much less efficient than the **SLS**-to-**SSOR** transformation. By less efficient we mean that the produced prove obligations are harder to prove. Nevertheless since we were able to verify the examples using a less efficient technique they also have to be verifiable by the **SLS**-to-**SSOR** transformation.

- The **SLS**-to-**SSOR** transformation is used to generate a pair of pre- and post-conditions which are used in the method specifications. On the other hand **WPC** just checks the consistency between an **SLS** term and the **SSOR** but does not produce an **SSOR** from an **SLS** term. To deal with this we had to explicitly provide pre- and post-conditions for some stereotype operations. These manually provided specifications can possibly differ from automatically generated once. It is possible that a human specification writer could perform some optimizations which are not feasible for an automatic theorem prover. We assume that this effect comes up only in complicated examples. In our case study the most complicated stereotype operations are inductive operations. For all these operations a specification writer has to explicitly provide an **SSOR** for the operation. In other words, the described effect comes up only in simple non-recursive operations and therefore should not affect the results of the case study significantly.

Another technical problem is related to limitations of the underlying automatic theorem prover. Even though our approach does not require inductive proofs and quantifier alternation, not all proof obligations can be verified automatically. To deal with it we use a standard **Boogie** approach to this problem: an intermediate assertion. This intermediate assertion is used by a theorem prover as a hint and facilitates verification of the required proof obligations. In our experiments we group intermediate assertions relevant to a specific subgoal and use a consistent notation to identify them. We mark the beginning and the end of the intermediate assertion as $//begin :: proof$ and $//end :: proof$, respectively. Most of the subgoals are used to prove rather simple properties. Nevertheless the subgoal descriptions are quite awkward and verbose. We believe that there are several reasons for this:

- The first one is that we use **WPC** but not the **SLS**-to-**SSOR** transformation. As we mentioned above this results in less efficient proof obligations. The intermediate subgoals make these proof obligations feasible for automatic verification.

- Another reason is the absence of a build in mechanism for handling user provided hints for splitting of subgoals in **Boogie** and the underlying automatic theorem prover. In the current approach the intermediate assertion is incorporated by **WPC** in the resulting proof obligation as a usual programm specification. We believe that they can be treated more efficiently. For instance, to prove that $A \Rightarrow B$ with the subgoal C it is enough to prove that $A \Rightarrow C$ and $C \Rightarrow B$. In this way we use C to prove that B holds and then immediately throw it away. By this we avoid polluting the rest of the proof obligations by C and make verification of B more directed.

One more technical problem which we have met was related to reusability. Since we use a direct **Boogie** encoding we cannot use specification operation calls in the methods specifications. Instead we have to explicitly duplicate stereotype operation specifications in the method specifications. As a result most method specifications are huge and look ugly. Nevertheless it is only a problem of the manual encoding. If we use use a specialized tool instead of the manual encoding then the method specifications would look exactly as we have specified them in this chapter.

The absence of a tool complicates the verification of the examples significantly. It requires many duplications of specifications, and the generated proof obligations are harder to verify. Due to this reason we postpone some of the universal transformation verifications until the tool is developed. Specifically we omit the verification of the universal transformation of the tree. Instead we have specified and verified only the inductive tree operations which are used in examples. Since we introduced $f_{\texttt{Desc}}$ only to deal with the universal transformation of the tree, we omit $f_{\texttt{Desc}}$ from the **Boogie** specification of the tree stereotype. We omit the specification of the instance identifiers, too.

The rest of the section is dedicated to the **Boogie** verification of specific examples. For each example we provide the verification time. The verification is done on a computer with i5 2.40GHz CPU and 4 GB of memory.

5.5.1 Composite design pattern

In this subsection we consider verification of the composite pattern which we have considered in the **Subsection 5.2**. The **Boogie** specification of the example consists of the following files:

- "Heap.bpl": contains definitions of the program heap, heap reference, and heap field. The heap is defined as a polymorphic map from a pair (reference, field) to the value of the field type. The file also contains the definition of the function $IsNotAllocated$ which checks whether a reference points to an object which is not allocated.
- "Region.bpl": contains the definition of regions and region operations. A region is a set of references. In **Boogie** we define regions as maps from references to boolean values. The file also contains several procedures which are used to specify the update of the fields of all objects which belong to a given region. We use these procedures to specify stereotype operations.
- "Auxiliary.bpl": contains specifications of auxiliary functions, e.g., maximum of two numbers, and of a set of numbers.
- The definition of the tree stereotype is provided in the following files:
  - "TreeStereotype.bpl": defines tree stereotype items. Here we define stereotype items as heap fields. In this way we simplify speci-

Name of the verified **Boogie** procedure	Verification time in seconds	Description of the procedure
removeSubTree	3.67	stereotype operation `removeSequenceRelation`
addSubTree	2.64	stereotype operation `addSequenceRelation`
creatEmptyTree	0.16	stereotype constructor
composite..ctor	0.13	*Composite* constructor
composite..update	0.71	*Composite* method update
composite..add	0.34	*Composite* method add child
composite..remove	0.39	*Composite* method remove child
main	0.52	client code
create_main	1.06	client code objects allocation

Figure 5.19: The results of the composite design pattern verification.

fications. We consider a more advanced specification of stereotype items in **Boogie** in **Subsection 5.5.4**.

- "TreeOperations.bpl": contains basic tree operations, a constructor, and the predicate which checks that a given predicate forms a singleton tree stereotype instance.
- "TreeInvariant.bpl": contains the definition of the tree stereotype invariants. The file contains both definition of **SysInvEl** and **SysInv** for the tree stereotype. As we have mentioned above we do not formalize stereotype instances and therefore omit the specification of **SysInvID**.

- "CompositeType.bpl": contains the definitions of the fields of the class *Composite* which are provided in Figure 5.6.
- "CompositeInvariant.bpl": contains the definition of behavioral and glue invariants of the class *Composite*. Here we do not check that a glue invariant has visible state semantics. Instead we have added the glue invariant to the behavioral invariant. We explicitly state when the combined invariant can be assumed and when its validity has to be verified.
- "InvCompositeAx.bpl": to facilitate the verification of the behavioral

Name of the verified **Boogie** procedure	verification time in seconds
ProveInvCompositeFrame	0.88
ProveInvCompositeStFrame	0.02
ProveInvCompositeRemoveChild	0.06
ProveInvCompositeAddChild	0.01
ProveInvCompositeUpdateChild	0.02
ProveInvCompositeSt2Composite	0.02

Figure 5.20: The results of the composite design pattern.

invariants we add several lemmas which describe properties of the behavioral invariants and the behavioral invariant states. The proofs of all lemmas do not require induction. Most of the proofs do not require any hints from the user. The proofs are provided in "InvCompositeProofs.bpl".

- "CompositeOperations.bpl": provides definitions and specifications of the methods of the class *Composite*.
- "client.bpl": contains client code which uses the *Composite* class. The client code allocates several objects of class *Composite*. Then the client code executes methods *add* and *remove* to establish relations between objects. It is verified that the resulting data structure preserves the behavioral and topological invariants. It is also verified that the field *childrenNumber* of objects of types *Composite* contains the correct number of descendants.

Together with the Boogie specification files we deliver two script files which can be used to start the verification process:

- "run.bat" starts the verification of the composite design patterns. The verification also covers stereotype operations and client code.
- "runInvComp.bat" verifies lemmas which describe properties of the behavioral invariants and behavioral invariant states.

The results of the verification are provided in the following xml files:

- "result.xml" contains the results of the verification. We present the result of the verification on Figure 5.19.
- "resultInvComp.xml" contains the results of verifying the lemmas. We present result on Figure 5.20.

5.5.2 PIP

In this subsection we consider the verification of the implementation of **PIP** which we have considered in **Subsection 5.4**. The Boogie specification of the example consists of the following files.

- "Heap.bpl", "Region.bpl", "Auxiliary.bpl", "TreeStereotype.bpl", "TreeOperations.bpl", and "TreeInvariant.bpl": are the same as in **Subsection 5.5.1**.
- "MSet.bpl": contains the definition of multisets and of relevant operations.
- "ListStereotype.bpl", "ListOperations.bpl", and "ListInvariant.bpl": contain definitions of the sequence stereotype and sequence stereotype operations. Here we use the simplified version of the sequence stereotype which is defined in Figure 4.2. The version is precise enough to verify the correctness of the **PIP** implementation. "ListOperations.bpl" contains the stereotype constructor *CreatEmptyList* and the stereotype operation *AddToEnd*. *AddToEnd* is a simplified version of `addTreeRelation` which assumes that the second merged stereotype instance is a singleton.
- "TreeToList.bpl": contains the definition of the stereotype operation *TreeToList*. The operation is used to verify method *acquire* of the **PIP** node. *TreeToList* is defined as the sequential composition of operations *TreeToListGetList* and *TreeToListRemTree* which are also defined in "TreeToList.bpl". *TreeToListGetList* uses a tree path to construct a sequence which contains objects which belongs to the path. The operation definition uses *AddToEnd*. *TreeToListRemTree* is an implementation of the operation `removePathTree` which is defined in Figure 4.29.
- "NodeType.bpl": the class contains the definition of the fields of the class *Node*. The class represents a **PIP** node which is defined in Figure 5.16. The file contains the following fields definitions:
  - *blockedBy* is defined as *Nbb*.
  - $defaultPriority$ is defined as *Ndpr*.
  - $currentPriority$ is defined as *Ncpr*.
  - $priorities$ is defined as *Npr*.
  - `st` is defined as *NSt*. The type of *NSt* is the enumeration which consists of the following three values *StF*, *StL*, *StD*. *StD* corresponds to $\texttt{st}_D$. $\texttt{st}_L$ is represented by two states *StF* and *StL*. *StL* corresponds to $\texttt{st}_L$ when *Nbb* of the object is not null. *StF* corresponds to $\texttt{st}_L$ when *Nbb* of the object is null.

Name of the verified **Boogie** procedure	Verification time in seconds	Description of the procedure
MSet.increment	0.98	addition of an element to a multiset
MSet.decrement	0.01	removal of an element from a multiset
CreatEmptyList	0.15	`createSingletonSequence` stereotype constructor
AddToEnd	6.29	a simplified version of `addSequenceRelation` stereotype operation
TreeToList	0.13	moving of a tree path to a sequence stereotype operation
TreeToListGetList	5.05	sequence construction from a tree path stereotype operation
TreeToListRemTree	5.01	`removePathTree` stereotype operation
Node..ctor	5.16	constructor of the **PIP** node
Node..updatePriorities	71.52	method *updatePriorities* of the **PIP** node
Node..release	7.23	method *release* of the **PIP** node
Node..acquire	77.71	method *acquire* of the **PIP** node
main	39.08	client code
main_create	6.09	client code objects allocation

Figure 5.21: The results of **PIP** verification.

Name of the verified **Boogie** procedure	verification time in seconds
ProveInvPrFrame1	1.09
ProveInvPrFrame2	0.03
ProveInvPrInvPrSt	0.15
ProveInvPrRemoveChild	0.07
ProveInvPrAddChild	0.06
ProveInvPrStFrame	0.03
ProveInvPrStFromUpdate	0.05
ProveInvPrStToUpdate	0.05
ProveInvPrStInvPr	0.05

Figure 5.22: The results of verifying the **PIP** lemmas.

- "NodeInvariant.bpl": Contains the definition of the behavioral and the glue invariant of the class *Node*.
- "InvPrAxioms.bpl": to facilitate verification of the behavioral invariants we add several lemmas which describe properties of the behavioral invariants and the behavioral invariant states. The proofs of all lemmas do not require induction. Most of the proofs do not require any hints from the user. The proofs of the lemmas are provided in "InvPrProofs.bpl".
- "Node.bpl": provides definitions and specifications of the methods of the class **PIP** *Node*.
- "client.bpl": contains client code which uses the **PIP** *Node* class. The client code allocates several objects of class **PIP** *Node*. Then the client code executes methods *acquire* and *release* to establish relations between the objects. It is verified that the resulting data structure preserves the behavioral and topological invariants. We also verify the precise heap topology of the resulting data structure.

Together with the Boogie specification files we deliver two script files which can be used to start the verification process:

- "run.bat" starts verification of the **PIP** implementation. The verification also covers stereotype operations and client code.
- "runInvPr.bat" verifies the lemmas which describe properties of the behavioral invariants and the behavioral invariant states.

The results of the verification are provided in the following xml files:

- ”result.xml” contains the results of the verification. We present the result of the verification in Figure 5.21. We omit the results of the tree stereotype operations verification. We have already considered them in **Subsection 5.5.2**.
- ”resultInvPr.xml” contains the results of verifying the lemmas. We present the results in Figure 5.22.

5.5.3 List with iterator

In this subsection we consider the verification of a simple list with iterator. The example is a simplified version of the list with iterator which is provided in **Section 5.1**. The version which we consider in this section is simplified in the following way:

- It uses only `Sequence` stereotype to specify that the list nodes form an acyclic list. Properties which are related to the `RelationInversion` stereotype are ignored.
- The list is singly-linked, not a doubly-linked.
- We drop the *value* field of the list node.

In this example we use the full version of the `Sequence` stereotype which is defined in Figure 4.5. We also use non standard basic operations. Instead of `addSequenceRelation` and `removeSequenceRelation` we use as basic operations addition of an element to a sequence and removal of an element from a sequence.

The **Boogie** specification of the example consists of the following files:

- ”Heap.bpl” and ”Region.bpl”: are the same as in **Subsection 5.5.1**.
- ”ListStereotype.bpl”, ”ListOperations.bpl”, and ”ListInvariant.bpl”: contain the definitions of the sequence stereotype and sequence stereotype operations. ”ListOperations.bpl” contains the following stereotype operations and constructor:
  - *CreatEmptyList* - sequence stereotype constructor.
  - *AddToEnd* - adds an element to the end of a sequence.
  - *InsertAtTheMiddleAfter* - adds an element into the middle of a sequence after the given element.
  - *InsertAfter* - adds an element into a sequence after the given element.
  - *RemoveToEnd* - removes the last element of a sequence.
  - *RemoveBegin* - removes the last element of a sequence.

Name of the verified **Boogie** procedure	Verification time in seconds	Description of the procedure
CreatEmptyList	0.98	stereotype constructor
AddToEnd	3.18	stereotype operation add elment to the end
InsertAtTheMiddleAfter	17.77	stereotype operation add elemnt to the middle
InsertAfter	0.08	stereotype operation add elemnet
RemoveEnd	4.45	stereotype operation remove the lsat element
RemoveBegin	4.37	stereotype operation remove the first element
RemoveFromTheMiddle	9.47	stereotype operation remove a middle element
RemoveEl	0.12	stereotype operation remove element
GetItBegin	0.09	method of the list get iterator
ItRemoveNext	1.20	method of iterator remove elment
ItInsertAfter	0.76	method of iterator add elment
main	0.28	client code

Figure 5.23: The results of verifying the list with iterator.

  - *RemoveFromTheMiddle* - removes an element from the middle of a sequence.
  - *RemoveEl* - removes an element from a sequence.

- "ListImplementation.bpl": contains the implementation of the list with iterator. The implementation consists of three classes, list node, list, and list iterator. The only field of the list node is *Node_next* which points to the next list node. The only field of the list is *List_head* which points to the list head. List iterator has two fields, *It_el* which refers to the list node and *It_List* which refers to the list to which *It_el* belongs. The file also specifies list and iterator invariants. The list invariant guarantees that list nodes form an acyclic singly-linked list. The list iterator invariant guarantees that if *It_el* is not null then the list node which is pointed to by *It_el* belongs to *It_List*.

  The file also contains client code which uses the list and the iterator. The client code uses iterators to transfer a list node from one list to another.

Together with the Boogie specification files we deliver the script files which can be used to start the verification process. "run.bat" starts verification of the list with the iterator. The verification also covers stereotype operations and client code.

The the results of the verification are provided in the file "result.xml". We present the results of the verification on Figure 5.23.

5.5.4 Universal transformation of the sequence stereotype

In this subsection we consider the verification of the universal transformations of the sequence stereotype. We discussed the transformation at the end of **Section 4.1**. The complete definition of the sequence universal transformations is provided in **Appendix A**. In this example we use the full version of the `Sequence` stereotype which is defined in Figure 4.5.

The **Boogie** specification of the universal transformations consists of the following files:

- "Heap.bpl" is the same as in **Subsection 5.5.1**.
- "Set.bpl": contains the definition of the type set and the set manipulating operations.
- "Region.bpl": contains the definition of the region type and region manipulating operations. We define the region type as a set of locations. The file also contains the definitions of maps from a reference to a reference and from a reference to a region.

Name of the verified **Boogie** procedure	Verification time in seconds	Description of the procedure
CreatSingletonSeq	0.94	stereotype constructor
AddSequenceRelation	30.89	stereotype operation `addSequenceRelation`
RemSequenceRelation	6.29	stereotype operation `removeSequenceRelation`
AddSetSequenceRelation	31.64	stereotype operation `addSetSequenceRelation`
RemSetSequenceRelation	52.00	stereotype operation `removeSetSequenceRelation`
SplitSeqOnSingletons	17.14	stereotype operation split a sequence
MergeSeqOfSingletons	3.15	stereotype operation merge a sequence
reverse	44.97	stereotype operation `reverseSequence`

Figure 5.24: The results of verifying the sequence universal transformations.

- ”Stereotype.bpl”: contains the definition of the stereotype item and stereotype slice. A stereotype slice is defined as a map from the stereotype item name to the value of the stereotype item type. The types of stereotype items are either maps from a reference to a region or maps from a reference to a region. The file also contains the definition of the *Elements* stereotype items and the predefined stereotype system invariant **SysInvEl**.
- ”sequenceSt.bpl”: contains the definition of the stereotype items and invariants of the `Sequence` stereotype.
- ”sequenceBasicOp.bpl”: contains definitions of the stereotype constructor and basic stereotype operations.
- ”sequenceIndOp.bpl”: contains definitions of inductive stereotype operations.
- ”reverse.bpl”: contains definitions of the following stereotype operations:
  - *SplitSeqOnSingletons* is the stereotype operation which splits a sequence into singleton sequences.
  - *MergeSeqOfSingletons* is the stereotype operation which merges a sequence of singleton sequences into a sequence.
  - *reverse* uses two of the above operations to define the reversal of a sequence.

Together with the Boogie specification files we deliver the script files which can be used to start the verification process. ”run.bat” starts verification of the universal transformations of the sequence stereotype.

The results of the verification are provided in the file ”result.xml”. We present the results on Figure 5.23.

Chapter 6

Stereotype operations

In this chapter we consider stereotype operations in detail. As we have seen in previous sections we use stereotype operations to change the values of stereotype items in a consistent way. We are presenting a methodology for the construction of stereotype operations which achieves the following goals:

- **Expressivity:** Our ultimate goal is to be able to express *an arbitrary computable transformation.* We achieve this by constructing a so-called *universal transformation* for a given stereotype. A universal transformation uses hints from a developer to compose basic stereotype operations so that they express a desired operation. We provide a language dedicated to the construction of stereotype operations, the specification language of specifications (abbreviated as **SLS**) in **Section 6.3**. The language has an operational nature and provides a natural way to construct stereotype operations. Applications of **SLS** are demonstrated in **Appendix A** and **Appendix B** where we construct universal transformations for the `Sequence` and `Tree` stereotypes respectively.

- **Provability:** The above goal can be easily achieved by using quantifier alternation and transitive closure. Nevertheless there is not much use for such specifications. The problem is that such specifications can be hardly verified automatically. Quite often the validity proof of the specification requires inductive reasoning and witness guessing. Both of them are problematic for automatic theorem provers. This is why we limit our approach to the verification of specifications which are universally quantified and do not contain transitive closures. The proof obligations which we generate to verify the specifications' validity have the form $\varphi \Rightarrow \phi$ where both φ and ϕ are universally quantified and do not contain transitive closures. As a result our proof obligations are feasible for automatic verification. The price which we pay for it is a high complexity of the definition of universal transformation. Luckily we have to provide this complicated specification only once per stereotype. As soon as we have a universal transformation we can express any other stereotype operation almost without additional effort.

- **Consistency validation and preservation of the stereotype invariant:** All constructed stereotype operations have to be consistent and preserve certain invariants which we describe below. We directly prove consistency and invariant preservation for basic operations. As we mentioned above all stereotype operations are constructed from basic operations. Using this observation we lift invariant preservation and consistency validation from basic operations to all other stereotype operations.

- **Expressivity vs. provability mismatch:** To overcome the mismatch we provide a translation from an arbitrary **SLS** term into a pair of pre/post conditions in the universally quantified fragment of **FOL**. We perform this translation in two steps. In the first step we transform an operation into a so-called standard stereotype operation representation (abbreviated as **SSOR**). This representation consists of a pre-condition, translation rules, frame rules, and a list of affected stereotype instances. The **SSOR** is defined in **Section 6.2**. The translation from an arbitrary **SLS** term to a corresponding **SSOR** is defined in **Section 6.4**. In the second step, which is rather trivial, we transform an **SSOR** into a pair of universally quantified pre/-post conditions, which establish a logical connection between states of the affected stereotype slice before and after an execution of the stereotype operation. We provide a soundness theorem for the **SLS**-to-**SSOR** translation. The soundness theorem guarantees that if the corresponding proof obligations hold then the **SLS**-to-**SSOR** translation preserves the operational semantics.

The rest of the chapter is organized in the following way.

In **Section 6.1** we provide a methodology for set description. The main goal of this methodology is to characterize sets with the help of characteristic formulas. For each set we have two characteristic formulas; the positive and the negative characteristic formulas. The positive characteristic formula can be used to check that a given element belongs to the set. The negative characteristic formula can be used to check that a given element does not belong to the set. Both the positive and the negative characteristic formulas have to be universally quantified and do not contain transitive closures. We also show how to reduce verification of some properties of validation of universally quantified formulas. The above constraints enable automatic verification by **SMT** theorem provers of set properties described in our methodology.

In **Section 6.2** we introduce the standard stereotype operation representation (denoted as **SSOR**). The **SSOR** of a stereotype operation describes the set of locations which are affected by the operation execution and the set of locations which are preserved by the operation execution. The **SSOR** also describes the new values which are assigned to the affected locations. We use the set methodology introduced in **Section 6.1** to describe **SSOR**s. Therefore **SSOR**s also can be handled by **SMT** theorem provers.

In **Section 6.2** we also describe the properties which we expect from **SSOR**s. For instance, we expect that **SSOR**s preserve the stereotype invariants and are consistent. We call the **SSOR**s for which the above property holds proper.

In **Section 6.3** we introduce the specification language of specifications (denote as **SLS**). We already considered **SLS** in **Subsection 3.3.2**. **SLS** is the language which is used to describe stereotype operations. In **Section 6.3** we define both the syntax and the semantics of **SLS**.

In **Section 6.4** we define the translation from an arbitrary **SLS** term to an **SSOR**. We also define the correspondence between **SLS** terms and **SSOR**s. An **SSOR** corresponds to an **SLS** term if and only if they update the affected stereotype slices in the same way. In **Section 6.4** we prove a soundness theorem. The soundness theorem guarantees that the translation of an arbitrary **SLS** to **SSOR** corresponds to the **SLS** term and is proper.

We conclude the chapter by **Section 6.5**. In **Section 6.5** we describe how **SLS** can be used to define the universal transformation for a practically important class of stereotypes. The universal transformation of a stereotype *St* is the most general stereotype operation for *St*. It can be used to define any other stereotype operation for *St*. In **Appendix A** and **Appendix B** we use the approach described in **Section 6.5** to define the universal transformations for the `Sequence` and `Tree` stereotypes.

6.1 Methodology for set description

Our experiments with **SMT** provers demonstrate that quite often they cannot handle existential quantification and quantifier alternation. Therefore our goal is to reduce stereotype specifications from **FOL** to the universally quantified fragment of **FOL**. In this section we consider how the specific set operations can be described in the universally quantified fragment of **FOL**. In the next section we use these set operations to specify stereotype operations.

We have a complete set description if for any given object we can prove whether the object belongs to the set or not. Usually if we can check the property $o \in S$ then we can express the property $o \notin S$ as $\neg(o \in S)$, and vice versa. Unfortunately it does not work if we limit ourselves to universally quantified formulas only. If the validity of $o \in S$ can be checked by a universally quantified formula φ then the validity of $o \notin S$ can be checked by $\neg\varphi$. However, $\neg\varphi$ is existentially quantified and therefore we cannot use it. Therefore we have to consider three different classes of sets represented by universally quantified formulas: sets for which we can check that an element belongs to the set, sets for which we can check that an element does not belong to a set, and sets for which we can check both.

By $\varphi[\mathbf{v}]$ we denote that the formula φ which among others depends on the variable $\mathbf{v}$. By $\varphi[\mathbf{v} := e]$ we denote the result of substituting of all occurrences of $\mathbf{v}$ by e.

Definition 24 (Positive set)**.** *A set S is positive if and only if there exists a universally quantified formula $\varphi^+_S[\boldsymbol{v}]$ such that for each quantifier-free expression e the following property holds: $e \in S \Leftrightarrow \varphi^+_S[\boldsymbol{v} := e]$.*

There is an alternative way to describe positive sets using existential quantifiers.

Lemma 2 (An alternative definition of positive set)**.** *A set S is positive if and only if there exists an existentially quantified formula $\psi^+_S[\boldsymbol{v}]$ such that for each quantifier-free expression e the following property holds: $e \notin S \Leftrightarrow \psi^+_S[\boldsymbol{v} := e]$.*

Proof. We can define ψ^+_S as $\neg\varphi^+_S$. Since φ^+_S is universally quantified if and only if S is positive and ψ^+_S is existentially quantified if and only if φ^+_S is universally quantified we conclude that ψ^+_S is existentially quantified if and only if S is positive. On the other hand $e \in S \Leftrightarrow \varphi^+_S[\mathbf{v} := e]$ implies $e \notin S \Leftrightarrow \neg\varphi^+_S[\mathbf{v} := e] \Leftrightarrow \psi^+_S[\mathbf{v} := e]$. □

Definition 25 (Negative set)**.** *A set S is negative if and only if there exists a universally quantified formula $\varphi^-_S[\boldsymbol{v}]$ such that for each quantifier-free expression e the following holds: $e \notin S \Leftrightarrow \varphi^-_S[\boldsymbol{v} := e]$.*

There is an alternative way to describe negative sets using existential quantifiers.

Lemma 3 (An alternative definition of negative set)**.** *A set S is negative if and only if there exists an existentially quantified formula $\psi^-_S[\boldsymbol{v}]$ such that for each quantifier-free expression e the following holds: $e \in S \Leftrightarrow \psi^-_S[\boldsymbol{v} := e]$.*

Proof. We can define ψ^-_S as $\neg\varphi^-_S$. Since φ^-_S is universally quantified if and only if S is negative and ψ^-_S is existentially quantified if and only if φ^-_S is universally quantified we conclude that ψ^-_S is existentially quantified if and only if S is negative. On the other hand $e \notin S \Leftrightarrow \varphi^-_S[\mathbf{v} := e]$ implies $e \in S \Leftrightarrow \neg\varphi^-_S[\mathbf{v} := e] \Leftrightarrow \psi^-_S[\mathbf{v} := e]$. □

For a set S we denote the universally quantified formulas which check membership and absence of membership of an element in the set as $\varphi_S^+[\mathbf{v}]$ and as $\varphi_S^-[\mathbf{v}]$, respectively. We call them positive and negative characteristic formulas of the set S. For a set S we denote the existentially quantified formulas which check membership and absence of membership of an element in the the set as $\varphi_{\overline{S}}^-[\mathbf{v}]$ and $\varphi_{\overline{S}}^+[\mathbf{v}]$, respectively. We call them negative and positive alternative characteristic formulas of the set S.

Definition 26 (Complete set)**.** *A set S is complete if and only if it is both positive and negative.*

Definition 27 (Complement of a set)**.** *We denote the complement of a set S as $\overline{S}$. We define it as $\overline{S} \overset{def}{=} \{o | o \notin S\}$.*

Definition 28 (Image of a set)**.** *The image of a subset $S \subseteq X$ under a function $f : X \to Y$ is the subset $f(S) \subseteq Y$ defined by $f(S) \overset{def}{=} \{o | \exists o' \in S : f[o'] = o\}$. Here we denote by $f[o']$ the image of the element o' under the function f. We treat in a special way functions from a set into a power set. If $f : X \to \mathbb{P}(Y)$ then $f(S) \subseteq Y$ is defined by $f(S) \overset{def}{=} \{o | \exists o' \in S : o \in f[o']\}$.*

Definition 29 (Universe)**.** *We denote the universal set as $\mathbb{U}$. Since all objects belong to the universe we can define its characteristic formulas as: $\varphi_{\mathbb{U}}^+ = \mathrm{T}$ and $\varphi_{\mathbb{U}}^- = \mathrm{F}$. Therefore we can conclude that the universe is a complete set.*

Let us consider several examples which illustrate the above definitions. If f is a unary function then:

- $\forall x : f[x] \neq \mathbf{v}$ is the positive characteristic formula of the set $\overline{f(\mathbb{U})}$. In other words $\varphi_{\overline{f(\mathbb{U})}}^+ = (\forall x : f[x] \neq \mathbf{v})$. $\mathbf{v} \in \overline{f(\mathbb{U})}$ if and only if $\forall x : f[x] \neq \mathbf{v}$ holds. $\mathbf{v} \notin \overline{f(\mathbb{U})}$ if and only if $\exists x : f[x] = \mathbf{v}$ holds. Since the last formula is existentially quantified we conclude that $\overline{f(\mathbb{U})}$ is not a negative and therefore not a complete set.

- $\forall x : f[x] \neq \mathbf{v}$ is the negative characteristic formula of the set $f(\mathbb{U})$. In other words $\varphi_{f(\mathbb{U})}^- = (\forall x : f[x] \neq \mathbf{v})$. $\mathbf{v} \notin f(\mathbb{U})$ if and only if $\forall x : f[x] \neq \mathbf{v}$ holds. $\mathbf{v} \in f(\mathbb{U})$ if and only if $\exists x : f[x] = \mathbf{v}$ holds. Since the last formula is existentially quantified we conclude that $f(\mathbb{U})$ is not a positive and therefore not a complete set.

- If S is the set of fix-points of f then $\varphi_S^+ = (f[\mathbf{v}] = \mathbf{v})$ and $\varphi_S^- = (f[\mathbf{v}] \neq \mathbf{v})$. Since we can construct both the positive and the negative characteristic formulas we conclude that S is a complete set.

To simplify dealing with characteristic formulas we assume that they all use the same variable $\mathbf{v}$ to represent an element for which we check set membership. Below we deal not only with sets but also with cartesian products of sets. When we deal with a set $S_1 \times S_2$ we use **v** to refer to an element of the first set and **v'** to refer to an element of the second set.

Let us now consider basic properties of positive, negative, and complete sets.

Lemma 4 (Basic properties of positive sets)**.** *The class of positive sets is closed under union, intersection, and intersection of an unbound number of sets.*

Proof. To prove that a set is positive or negative it is enough to construct a corresponding characteristic formula.

- For any positive sets S_1 and S_2 an element described by an expression e belongs to $S_1 \cup S_2$ if and only if $(\varphi^+_{S_1} \vee \varphi^+_{S_2})[\mathbf{v} := e]$. Since this formula is universally quantified we can use it to define the characteristic function of $S_1 \cup S_2$: $\varphi^+_{S_1 \cup S_2} = \varphi^+_{S_1} \vee \varphi^+_{S_2}$
- in a similar way we can conclude that $\varphi^+_{S_1 \cap S_2} = \varphi^+_{S_1} \wedge \varphi^+_{S_2}$
- and that $\varphi^+_{\bigcap\limits_{i \in I} S_i} = \forall i \in I : \varphi^+_{S_i}$, where I is a set of indexes.

□

Lemma 5 (Basic properties of negative sets)**.** *The class of negative sets is closed under union, intersection, union of an unbounded number of sets, and image of a set. For image of a set S under function $f : X \to \mathbb{P}(Y)$ from a set into a power set we have the following extra requirement: for each $o \in X$, $f[o]$ is a negative set*

Proof. The proof is similar to **Lemma 4**.

- $\varphi^-_{S_1 \cup S_2} = \varphi^-_{S_1} \wedge \varphi^-_{S_2}$
- $\varphi^-_{S_1 \cap S_2} = \varphi^-_{S_1} \vee \varphi^-_{S_2}$
- $\varphi^-_{\bigcup\limits_{i \in I} S_i} = \forall i \in I : \varphi^-_{S_i}$, where I is a set of indexes.
- An element **v** doesn't belong to an image of a set $S \subseteq X$ under a function $f : X \to Y$ if and only if $\forall o \in X : o \in S \Rightarrow f[o] \neq \mathbf{v}$. From this we infer that the characteristic formula of an image of a set can be defined as $\varphi^-_{f(S)} = \forall o \in X : \varphi^-_S[\mathbf{v} := o] \vee f[o] \neq \mathbf{v}$. This formula is universally quantified.

- An element $\mathbf{v}$ doesn't belong to an image of a set $S \subseteq X$ under a function $f : X \rightarrow \mathbb{P}(Y)$ if and only if $\forall o \in X : o \in S \Rightarrow \mathbf{v} \notin f[o]$. From this we infer that the characteristic formula of an image of a set can be defined as $\varphi^{-}_{f(S)} = \forall o \in X : \varphi^{-}_{S}[\mathbf{v} := o] \vee \mathbf{v} \notin f[o]$. Since for each $o \in X$, $f[o]$ is a negative set we conclude that $\varphi^{-}_{f(S)}$ is universally quantified.

□

Later on we will see that the image of a negative set is crucial for the specification of heap updates. Essentially it is because we use the image of a set to construct the set of objects reachable via a field or a transitive closure of a field starting from an initial set of objects. Since this construction is so widely used we introduce an alternative, more convenient notation for it.

Definition 30 (Notations for the image of a negative set)**.** *We use the following abbreviations for the image of a negative set:*

- *By $S.f$ we denote $f(S)$, where $S : \mathrm{Reg}$ and $f : \mathrm{ref} \rightarrow \mathrm{ref}$ or $f : \mathrm{ref} \rightarrow \mathrm{Reg}$.*
- *By $S.it$ we denote $f(S)$, where $S : \mathrm{Reg}$, "it" is a stereotype item of a reference or a region type, and f is defined as $\forall o : f[o] = o.it$.*
- *By $S.g[exp]$ we denote $f(S)$, where $S : \mathrm{Reg}$, g is a function or a stereotype item from ref^2 into ref or Reg, and $\forall o : f[o] = g[o, exp]$.*
- *From **Lemma 5** we can see that to construct the characteristic formula of $S.f$ for a negative set we have to introduce a new quantified variable. In some cases we need to know the name of this variable to refer to it from other expressions. We use the notation $S\langle o\rangle.f$ to specify that the name of the newly introduced quantified variable is o.*

Lemma 6 (Relation between positive and negative sets)**.** *The complement of a positive set is a negative set, and vice versa.*

Proof.

- $\forall o : o \in \overline{S} \Leftrightarrow o \notin S$ implies $\varphi^{+}_{\overline{S}} = \varphi^{-}_{S}$
- $\forall o : o \notin \overline{S} \Leftrightarrow o \in S$ implies $\varphi^{-}_{\overline{S}} = \varphi^{+}_{S}$

□

Lemma 7 (Basic properties of complete sets)**.** *The class of complete sets is closed under union, intersection, negation, and complement.*

Proof. Since we define complete sets as the intersection of the classes of positive and negative sets, we immediately get the desired properties of union and intersection from **Lemma 4** and **Lemma 5**.

Since a complete set S is both positive and negative, we get from **Lemma 6** that $\overline{S}$ also is both positive and negative, which implies that $\overline{S}$ is complete. $\square$

Lemma 8 (Sets equality). *For a set A and a set B, if φ_B^+ is a positive characteristic function of A (1) then $A = B$.*

Proof. $\forall o : o \in A \overset{(1)}{\Leftrightarrow} \varphi_B^+[\mathbf{v} := o] \Leftrightarrow o \in B$ which implies $A = B$. $\square$

Unfortunately complete sets lack some properties which we need for our methodology. For instance, checks of membership in a union of an unbounded number of sets or in an image of a set require usage of an existential quantifier. We address these problems by requiring witness functions for such problematic operations. Essentially witness functions explicitly provide values for existentially quantified variables, which can be used to construct these operations.

Definition 31 (Union of an unbounded number of sets with a witness function). *For a complete set S, complete sets $S_i \subseteq S$, where $i \in I$, and a function $f : Ind \to S$, where Ind is a universe of indexes, we denote the union of S_i with a witness function f as $\overset{f}{\bigcup\limits_{i \in I}} S_i$. We characterize $\overset{f}{\bigcup\limits_{i \in I}} S_i$ by the following formulas: $\varphi^+_{\overset{f}{\bigcup\limits_{i\in I}} S_i} = \varphi_S^+ \wedge f[\boldsymbol{v}] \in I$ and $\varphi^-_{\overset{f}{\bigcup\limits_{i\in I}} S_i} = \varphi_S^- \vee f[\boldsymbol{v}] \notin I$.*

Lemma 9 (Semantics of a union of an unbound number of sets with a witness function). *For a complete set S, complete sets $S_i \subseteq S$, where $i \in I$, and a witness function $f : Ind \to S$ if*

- $\forall o \in S : f[o] \in I \Rightarrow o \in S_{f[o]}$ (1) *and*
- $\forall o \in S, i \in I : o \in S_i \Rightarrow f[o] \in I$ (2)

then $\overset{f}{\bigcup\limits_{i \in I}} S_i = \bigcup\limits_{i \in I} S_i$.

Proof. According to **Lemma 8** to prove $\overset{f}{\bigcup\limits_{i \in I}} S_i = \bigcup\limits_{i \in I} S_i$ it is enough to prove that $\varphi^+_{\overset{f}{\bigcup\limits_{i\in I}} S_i}$ is a positive characteristic function of $\bigcup\limits_{i \in I} S_i$.

We can easily see that (2) implies $\forall o \in S : (\exists i \in I : o \in S_i) \Rightarrow f[o] \in I (3)$.

$\forall o \in S : o \in \bigcup\limits_{i \in I} S_i \Rightarrow \exists i \in I : o \in S_i \overset{(3)}{\Rightarrow} f[o] \in I \Rightarrow \left(\varphi^{+}{}_{\substack{f \\ \bigcup\limits_{i \in I} S_i}} \right) [\mathbf{v} := o]$ (4)

$\forall o \in S : \left(\varphi^{+}{}_{\substack{f \\ \bigcup\limits_{i \in I} S_i}} \right) [\mathbf{v} := o] \Rightarrow f[o] \in I \overset{(1)}{\Rightarrow} o \in S_{f[o]} \wedge f[o] \in I \Rightarrow$
$\exists i \in I : o \in S_i \Rightarrow o \in \bigcup\limits_{i \in I} S_i$ (5)

From (4) $\wedge$ (5) we can conclude that $\forall o \in S : o \in \bigcup\limits_{i \in I} S_i \Leftrightarrow$
$\left(\varphi^{+}{}_{\substack{f \\ \bigcup\limits_{i \in I} S_i}} \right) [\mathbf{v} := o]$ (6).

On the other hand $\forall o : o \notin S \Rightarrow (\forall i \in I : o \notin S_i) \Rightarrow \bigcup\limits_{i \in I} S_i$ (7) and
$\forall o : o \notin S \Rightarrow \neg \varphi^{+}_{S} [\mathbf{v} := o] \Rightarrow \neg \left(\varphi^{+}_{\bigcup\limits_{i \in I} S_i} \right) [\mathbf{v} := o]$ (8). (7) $\wedge$ (8) implies
$\forall o \notin S : o \in \bigcup\limits_{i \in I} S_i \Leftrightarrow \left(\varphi^{+}{}_{\substack{f \\ \bigcup\limits_{i \in I} S_i}} \right) [\mathbf{v} := o]$ (9).

(6) $\wedge$ (9) implies $\forall o : o \in \bigcup\limits_{i \in I} S_i \Leftrightarrow \left(\varphi^{+}{}_{\substack{f \\ \bigcup\limits_{i \in I} S_i}} \right) [\mathbf{v} := o]$ (10), which proves that $\varphi^{+}{}_{\substack{f \\ \bigcup\limits_{i \in I} S_i}}$ is a positive characteristic formula of $\bigcup\limits_{i \in I} S_i$.

$\square$

Since both assumptions (1) and (2) from **Lemma 9** are universally quantified we can now prove properties of a union of an unbound number of sets using only universal quantifiers.

Every time we use a union of an unbounded number of sets with a witness function in a specification we expect that the assumptions (1) and (2) of the witness function hold. To avoid an unnecessary redundancy in specifications every time we use a union of an unbound number of sets with a witness function in the stereotype operation pre-condition we implicitly add assumptions (1) and (2) to the pre-condition of the operation. In similar way if a stereotype invariant mentions a union of an unbounded number of sets with a witness function we implicitly add assumptions (1) and (2) to the stereotype invariant.

We use **Lemma 9** to reason about the union of an unbound number of sets of references which participate in a stereotype slice *StName*. To use

Lemma 9 we have to define the set S for the stereotype slice *StName*. We define S as $\mathbf{Dom}(\mathit{StName})$ (see **Definition 4**). **Lemma 9** requires the set S to be complete. We prove the completeness of $\mathbf{Dom}(\mathit{StName})$ in the following lemma.

Lemma 10 ($\mathbf{Dom}(\mathit{StName})$ is a complete set)**.** *At each program point of the program which satisfies the syntactic constraints from **Definition 20** for each stereotype slice* StSlice *which is declared by the program, **Dom**(*StName*) is a complete set.*

Proof. To prove the completeness of $\mathbf{Dom}(\mathit{StName})$ it is enough to construct the positive and negative characteristic functions of $\mathbf{Dom}(\mathit{StName})$ at an arbitrary program point. Let us denote the heap at the program point by h. According to **Theorem 1**, $\mathbf{SysInvPart}[\mathit{StName}, h]$ has to hold at every program point. According to **Definition 18**, $\mathbf{SysInvPart}[\mathit{StName}, h]$ implies $\forall o : o \in \mathbf{Dom}(\mathit{StName}) \Leftrightarrow \mathtt{isAllocated}(h, o) \wedge$ $\mathtt{partIn}(\mathtt{typeOf}(h, o), \mathit{StName})$. Therefore we can define $\varphi^{+}_{\mathbf{Dom}(\mathit{StName})}$ as $\mathtt{isAllocated}(h, \mathbf{v}) \wedge \mathtt{partIn}(\mathtt{typeOf}(h, o), \mathit{StName})$ and $\varphi^{-}_{\mathbf{Dom}(\mathit{StName})}$ as $\neg\mathtt{isAllocated}(h, \mathbf{v}) \vee \neg\mathtt{partIn}(\mathtt{typeOf}(h, o), \mathit{StName})$. □

One very important application of **Lemma 9** is reasoning about the union of elements of a set of stereotype instances.

Definition 32 (Union of elements of a set of stereotype instances)**.** ***InstEl*** *:* Reg $\rightarrow$ Reg *is the function which maps a set of stereotype instances to the set of elements of these instances. We define **InstEl** in the following way. For a set of objects Inst,* $\mathbf{\mathit{InstEl}}(\mathit{Inst}) = \bigcup^{f_{\mathit{instID}}}_{i \in \mathit{Inst}} i.\mathrm{Elements}$*, where* $\forall o : f_{\mathit{instID}}[o] = i.\mathit{instID}$.

Lemma 11 (Semantics of union of elements of a set of stereotype instances)**.** *For each stereotype slice* StSlice *and set* Inst *if* $\mathit{Inst} \subseteq \mathbf{\mathit{Dom}}(\mathrm{StName})$, $\forall i \in \mathit{Inst} : i = i.\mathit{instID}$*, and* $\mathbf{\mathit{SysInv}}[\mathrm{StName}]$ *then the assumptions* (1) *and* (2) *from **Lemma 9** hold for the function* f_{instID} *and therefore* $\mathbf{\mathit{InstEl}}(\mathit{Inst}) = \bigcup_{i \in \mathit{Inst}} i.\mathrm{Elements}$

Proof. We would like to prove:

- $\forall o \in S : f[o] \in I \Rightarrow o \in S_{f[o]}$ (1) and
- $\forall o \in S, i \in I : o \in S_i \Rightarrow f[o] \in I$ (2)

where:

- $I = \mathit{Inst}$
- $S_i = i.\mathit{Elements}$

- $\forall o \in S : f[o] = o.\mathtt{instID}$
- $S = \mathbf{Dom}(\mathit{StName})$

Let us prove (1). (1) is equivalent to $\forall o \in \mathbf{Dom}(\mathit{StName}) : o.\mathtt{instID} \in \mathit{Inst} \Rightarrow o \in o.\mathtt{instID}.\mathit{Elements})$. We will prove the stronger property $\forall o \in \mathbf{Dom}(\mathit{StName}) : o \in o.\mathtt{instID}.\mathit{Elements})$ which implies validity of (1). Let us prove that for each $o \in \mathbf{Dom}(\mathit{StName})$, $o \in o.\mathtt{instID}.\mathit{Elements})$ (3) holds.

$\mathbf{SysInv}[\mathit{StName}]$ implies $o.\mathtt{instID} \in o.\mathit{Elements}$ and $o.\mathtt{instID} \in o.\mathtt{instID}.\mathit{Elements}$. Therefore we can conclude $\neg(o.\mathit{Elements} \sharp o.\mathtt{instID}.\mathit{Elements})$ (4).

On the other hand $\mathbf{SysInv}[\mathit{StName}]$ implies $(o.\mathit{Elements} \sharp o.\mathtt{instID}.\mathit{Elements}) \vee o.\mathit{Elements} = o.\mathtt{instID}.\mathit{Elements}$. Together with (4) it implies $o.\mathit{Elements} = o.\mathtt{instID}.\mathit{Elements}$ (5).

$\mathbf{SysInv}[\mathit{StName}]$ implies $o \in o.\mathit{Elements}$. In combination with (5) it implies (3).

Let us prove (2). (2) is equivalent to the following statement. For each $o \in \mathbf{Dom}(\mathit{StName})$ and $i \in \mathit{Inst}$ $o \in i.\mathit{Elements}$ implies $o.\mathtt{instID} \in \mathit{Inst}$. According to $\mathbf{SysInv}[\mathit{StName}]$, $o \in i.\mathit{Elements}$ implies $o.\mathtt{instID} = i.\mathtt{instID}$. Since $i \in \mathit{Inst}$ we can conclude that $i.\mathtt{instID} = i$ and therefore $o.\mathtt{instID} = i$. Since $i \in \mathit{Inst}$ and $o.\mathtt{instID} = i$, we have $o.\mathtt{instID} \in \mathit{Inst}$.

□

Let us now consider an image of a set with a witness function.

Definition 33 (Image of a set with a witness function)**.** *For complete sets A and S, and a function $f : A \to S$ we denote the image of the set A under the function f with a witness function $f^{-1} : S \to A$ as $\overset{f^{-1}}{f}(A)$. We characterize $\overset{f^{-1}}{f}(A)$ by the following formulas: $\varphi^{+}_{\overset{f^{-1}}{f}(A)} = \varphi^{+}_{S} \wedge \varphi^{+}_{A}[\boldsymbol{v} := f^{-1}[\boldsymbol{v}]]$ and $\varphi^{-}_{\overset{f^{-1}}{f}(A)} = \varphi^{-}_{S} \vee \varphi^{-}_{A}[\boldsymbol{v} := f^{-1}[\boldsymbol{v}]]$*

Lemma 12 (Semantics of an image of a set with a witness function)**.** *For complete sets A and S, and a function $f : A \to S$, and witness function $f^{-1} : S \to A$ if*

- *$\forall x \in A : f^{-1}[f[x]] = x$ (1) and*
- *$\forall x, y \in S : f^{-1}[a] = f^{-1}[b] \Rightarrow a = b$ (2)*

then $\overset{f^{-1}}{f}(S) = f(S)$.

Proof. According to **Lemma 8** to prove that $\overset{f^{-1}}{f}(A) = f(A)$ it is enough to prove that $\varphi^{+}_{\overset{f^{-1}}{f}(A)}$ is a positive characteristic function of $f(A)$.

For each $x \in S$, $x \in f(A) \Leftrightarrow \exists y : y \in A \wedge x = f[y] \overset{(2)}{\Leftrightarrow} \exists y : y \in A \wedge f^{-1}[x] = f^{-1}[f[y]] \overset{(1)}{\Leftrightarrow} \exists y : y \in A \wedge f^{-1}[x] = y \Leftrightarrow f^{-1}[x] \in A$. On the other hand if $x \notin S$ then, since the function f maps to the set S, $x \notin f(A)$. Therefore, for each x, $x \in f(A) \Leftrightarrow x \in S \wedge f^{-1}[x] \in A \Leftrightarrow \varphi^+_S[\mathbf{v} := x] \wedge \varphi^+_A[\mathbf{v} := f^{-1}[x]] \Leftrightarrow \varphi^+_{\substack{f^{-1}\\f(A)}}[\mathbf{v} := x]$, which proves that $\varphi^+_{\substack{f^{-1}\\f(A)}}$ is the positive characteristic formula of $f(A)$.

□

Similarly to the union of an unbounded number of sets with a witness function every time when we use an image of a set with a witness function in the stereotype operation pre-condition we implicitly add assumptions (1) and (2) to the pre-condition of the operation. In the similar way if a stereotype invariant mentions the image of a set with a witness function we implicitly add assumptions (1) and (2) to the stereotype invariant.

We use **Lemma 12** to reason about the images of sets of references which participate in a stereotype slice *StName*. To use **Lemma 12** we have to define the set S for the stereotype slice *StName*. Similarly to the union of an unbounded number of sets with a witness function we define S as **Dom**(*StName*) (see **Definition 4**).

By this we conclude the construction of set descriptions in our methodology. It is obvious that we need not only set descriptions, but also a way to check their properties. Like in the rest of our methodology we would like to check these properties using only universally quantified formulas. Let us prove several lemmas which we use to achieve this goal.

Lemma 13 (Check of subset relation)**.** *For a positive set S^+ and a negative set S^- we can check $S^- \subseteq S^+$ using universally quantified formulas only.*

Proof. $S^- \subseteq S^+$ holds if and only if $\forall o : o \in S^- \Rightarrow o \in S^+$. Which is equivalent to $\forall o : o \notin S^- \vee o \in S^+$. By using characteristic formulas we can reformulate this statement as $\forall o : \varphi^-_{S^-}[\mathbf{v} := o] \vee \varphi^+_{S^+}[\mathbf{v} := o]$, which is universally quantified and checks the desired property. □

Lemma 14 (Check of set disjointness)**.** *For a pair of negative sets S_1 and S_2 we can check $S_1 \sharp S_2$ using universally quantified formulas only.*

Proof. $S_1 \sharp S_2 \Leftrightarrow \forall o : o \notin S_1 \vee o \notin S_2 \Leftrightarrow \forall o : \varphi^-_{S_1}[\mathbf{v} := o] \vee \varphi^-_{S_2}[\mathbf{v} := o]$
The last statement is universally quantified and checks the desired property.

□

Unfortunately not all the properties can be expressed so easily. Some of them require existential quantifiers. One such property is the check of strict set inclusion. To check that we have to verify the existence of an element that belongs to one set but not to the other. To avoid the undesired existential quantification we use a witness element.

Definition 34 (Check of strict subset inclusion with a witness element)**.** *For a positive set S^+, a negative set S^-, and an element a we define $S^- \overset{a}{\subset} S^+$ as $(\forall o : \varphi^-_{S^-}[\boldsymbol{v} := o] \vee \varphi^+_{S^+}[\boldsymbol{v} := o]) \wedge \varphi^-_{S^-}[\boldsymbol{v} := a] \wedge \varphi^+_{S^+}[\boldsymbol{v} := a]$.*

Lemma 15 (Semantics of check of strict subset inclusion with a witness element)**.** *For a positive set S^+ and a negative set S^- $S^- \subset S^+$ holds if and only if there exists an element a such that $S^- \overset{a}{\subset} S^+$ holds.*

Proof. Let us assume that $S^- \overset{a}{\subset} S^+$ holds. $(\forall o : \varphi^-_{S^-}[\mathbf{v} := o] \vee \varphi^+_{S^+}[\mathbf{v} := o])$ implies $S^- \subseteq S^+$. On the other hand $\varphi^-_{S^-}[\mathbf{v} := a] \wedge \varphi^+_{S^+}[\mathbf{v} := a]$, which is equivalent to $a \notin S^- \wedge a \in S^+$, implies $S^- \neq S^+$. If we put these two properties together we get the desired property $S^- \subseteq S^+ \wedge S^- \neq S^+ \Rightarrow S^- \subset S^+$.

Let us now prove the desired property in the other direction. Let us assume that $S^- \subset S^+$. It implies $(\forall o : \varphi^-_{S^-}[\mathbf{v} := o] \vee \varphi^+_{S^+}[\mathbf{v} := o])$. On other hand $S^- \subset S^+$ implies $S^+ \setminus S^- \neq \varnothing$. Let us take as a an arbitrary element from $S^+ \setminus S^-$. It is obvious that for a we have $\varphi^-_{S^-}[\mathbf{v} := a] \wedge \varphi^+_{S^+}[\mathbf{v} := a]$, which concludes the proof. $\square$

The class of complete sets is a subset of both the class of positive and the class of negative sets. This implies that if a property holds for a positive or a negative set then the same property holds for a complete set.This observation extends the application of lemmas about positive an negative sets to the class of complete sets.

6.2 Standard stereotype operation representation

Now we can use our methodology of set description to describe transformation and frame rules. Later on we use these rules as building blocks of a standard stereotype operation representation.

Standard stereotype operation representations describe a relation between a new and an old stereotype slice. We call the variables which correspond to an old and a new slice **sOld** and **sNew**, respectively. We have already used the variable **sOld** to denote a stereotype slice in set expressions.

To simplify dealing with expressions we assume that they all use the same variable **sOld** to represent a stereotype slice whose properties are described by the expression.

Definition 35 (Transformation rule)**.** *A transformation rule is a triple $\langle \psi^-_S, it, val \rangle$. Where:*

- *"it" is a stereotype item whose values are changed by the rule*

- ψ_S^- *is an existential description of a negative set of objects whose values of the stereotype item "it" are affected by the transformation rule. The only free variables of* ψ_S^- *are* $\boldsymbol{v}$ *and the stereotype operation's input parameters.*

- *val is an expression which computes new values for stereotype item "it" of an object* $\boldsymbol{v}$*. The only free variables of val are* $\boldsymbol{v}$*, the stereotype operation's input parameters, and existentially quantified variables introduced by* ψ_S^-*.*

Definition 36 (Logical characterization of a transformation rule)**.** *A logical characterization of a transformation rule* $\langle\psi_S^-, it, val\rangle$*, denoted as* $[\![\langle\psi_S^-, it, val\rangle]\!]$*, establishes a relation between values of the stereotype item "it" of objects from set S in an old stereotype slice* ***sOld*** *and in a new stereotype slice* ***sNew****. We define it formally as*
$[\![\langle\psi_S^-, it, val\rangle]\!] = \big(\forall \boldsymbol{v} : (\neg\psi_S^-) \vee \boldsymbol{sNew.v}.it = val\big)$*.*

Definition 37 (Frame rule)**.** *A frame rule is a pair* $\langle\psi_S^-, it\rangle$*, where:*

- *"it" is a stereotype item whose values are preserved by the rule.*

- ψ_S^- *is an existential description of a negative set of objects whose values of the stereotype item "it" are preserved by the frame rule. The only free variables of* ψ_S^- *are* $\boldsymbol{v}$ *and the stereotype operation's input parameters.*

Definition 38 (Logical characterization of a frame rule)**.** *A logical characterization of a frame rule* $\langle\psi_S^-, it\rangle$*, denoted as* $[\![\langle\psi_S^-, it\rangle]\!]$*, states that values of the stereotype item "it" of objects from set S in an old stereotype slice* ***sOld*** *and in a new stereotype slice* ***sNew*** *are equal. We define it formally as* $[\![\langle\psi_S^-, it\rangle]\!] = \big(\forall \boldsymbol{v} : (\neg\psi_S^-) \vee \boldsymbol{sNew.v}.it = \boldsymbol{sOld.v}.it\big)$*.*

We introduce a special notation for frame rules which states that values of all stereotype items are preserved for certain objects. We denote such frame rules as $\langle\psi_S^-, *\rangle$.
$[\![\langle\psi_S^-, *\rangle]\!] = \big(\forall \mathbf{v}, it : (\neg\psi_S^-) \vee \mathbf{sNew}.\mathbf{v}.it = val\big)$.

If we deal with a stereotype item of a functional type then the first parameter of a transformation or a frame rule describes a set of pairs of affected objects and input parameters of the stereotype item, not just a set of objects. By this we achieve better granularity of the transformation and the frame rules. To highlight this granularity we denote a characteristic formula ψ_S^- of a set of pairs S as $\psi_{S_1}^- \times \psi_{S_2}^-$, where $S = S_1 \times S_2$. Using this notation we refine the logical characterization of a transformation and a frame rule for stereotype items of a functional type:

- $[\![\langle\psi_{S_1}^- \times \psi_{S_2}^-, it, val\rangle]\!] =$
$\Big(\forall \mathbf{v}, \mathbf{v'} : (\neg\psi_{S_1}^-) \vee (\neg\psi_{S_2}^-) \vee \mathbf{sNew}.\mathbf{v}.it[\mathbf{v'}] = val\Big)$.

- $[\![\langle\langle\psi^-_{S_1} \times \psi^-_{S_2}, it\rangle]\!] =$
  $\Big(\forall \mathbf{v}, \mathbf{v'} : (\neg\psi^-_{S_1}) \vee (\neg\psi^-_{S_2}) \vee \mathbf{sNew}.\mathbf{v}.it[\mathbf{v'}] = \mathbf{sOld}.\mathbf{v}.it[\mathbf{v'}]\Big)$.

Quite often we use set expressions to represent the first parameter of a transformation or a frame rule. By this we mean that a rule affects all values which belong to the set which is mentioned as the first parameter of the rule. In other words:

- $\langle S, it, val\rangle$ is an abbreviation for $\langle \mathbf{v} \in S, it, val\rangle$.
- $\langle S, it\rangle$ is an abbreviation for $\langle \mathbf{v} \in S, it\rangle$.
- $\langle S_1 \times S_2, it, val\rangle$ is an abbreviation for $\langle (\mathbf{v} \in S_1) \times (\mathbf{v'} \in S_2), it, val\rangle$.
- $\langle S_1 \times S_2, it\rangle$ is an abbreviation for $\langle (\mathbf{v} \in S_1) \times (\mathbf{v'} \in S_2), it\rangle$.

Now we can use transformation and frame rules to introduce a standard stereotype operation representation. The representation is used to define a logical specification of stereotype operations.

Definition 39 (Standard stereotype operation representation)**.** *A standard stereotype operation representation (abbreviated as **SSOR**) of a stereotype operation* $\boldsymbol{op}(v_1 : T_1, \ldots, v_n : T_n)$*, where* $v_1, \ldots, v_n$ *are input parameters and* $T_1, \ldots, T_n$ *are their types, is a triple of:*

- *a universally quantified pre-condition* $\boldsymbol{pre_{op}}$
- *a list of transformation rules* $\langle\psi^-_{S^t_1}, it^t_1, val_1\rangle, \ldots, \langle\psi^-_{S^t_n}, it^t_n, val_n\rangle$
- *a list of frame rules* $\langle\psi^-_{S^f_1}, it^f\rangle, \ldots, \langle\psi^-_{S^f_m}, it^f_m\rangle$

Definition 40 (Logical characterization of an **SSOR**)**.** *An **SSOR** of* $\boldsymbol{op}(v_1 : T_1, \ldots, v_n : T_n)$ *can be logically characterized by a pair of pre and post conditions* $[\![\boldsymbol{SSOR_{op}}]\!] = \langle \boldsymbol{pre_{op}}, \boldsymbol{post_{op}}\rangle$*. Here* $\boldsymbol{pre}_{op}$ *can be immediately extracted from the **SSOR** and* $\boldsymbol{post_{op}} = \Big(\bigwedge_{i=1}^{n} [\![\langle\psi^-_{S^t_i}, it^t_i, val^t_i\rangle]\!]\Big) \wedge \Big(\bigwedge_{i=1}^{m} [\![\langle\psi^-_{S^f_i}, it^f_i\rangle]\!]\Big)$*. We can see that both* $\boldsymbol{pre_{op}}$ *and* $\boldsymbol{post_{op}}$ *are universally quantified.* $\boldsymbol{pre_{op}}$ *may depend on free variables* $v_1, \ldots, v_n$ *and **sOld**.* $\boldsymbol{post_{op}}$ *may depend on the same free variables plus **sNew**.*

Let us consider an example of a transformation rule. In **Section 4.1** on Figure 4.6 we defined the operation `addSequenceRelation`$\langle$`Sequence`$\rangle$$(o : \mathit{ref!}, o' : \mathit{ref!})$. The **SSOR** of the stereotype operation `addSequenceRelation` is:

- the operation precondition $\mathbf{pre}_{\texttt{addSequenceRelation}}$ is $\mathbf{sOld}.o.\texttt{instID} \neq \mathbf{sOld}.o'.\texttt{instID}$.

- the list of transformation rules of `addSequenceRelation` is

  - $\langle \mathbf{v} = \mathbf{sOld}.o.\texttt{last}, \texttt{next}, \mathbf{sOld}.o'.\texttt{first}\rangle$
  - $\langle \mathbf{v} = \mathbf{sOld}.o'.\texttt{first}, \texttt{prev}, := \mathbf{sOld}.o.\texttt{last}\rangle$
  - $\langle \mathbf{v} \in \mathbf{sOld}.o.\mathit{Elements} \cup \mathbf{sOld}.o'.\mathit{Elements}, \texttt{first}, \mathbf{sOld}.o.\texttt{first}\rangle$
  - $\langle \mathbf{v} \in \mathbf{sOld}.o.\mathit{Elements} \cup \mathbf{sOld}.o'.\mathit{Elements}, \texttt{last}, \mathbf{sOld}.o'.\texttt{last}\rangle$
  - $\langle \mathbf{v} \in \mathbf{sOld}.o.\mathit{Elements}, \texttt{Next}^*, \mathbf{sOld}.\mathbf{v}.\texttt{Next}^* \cup \mathbf{sOld}.o'.\mathit{Elements}\rangle$
  - $\langle \mathbf{v} \in \mathbf{sOld}.o'.\mathit{Elements}, \texttt{Prev}^*, \mathbf{sOld}.\mathbf{v}.\texttt{Prev}^* \cup \mathbf{sOld}.o.\mathit{Elements}\rangle$
  - $\langle \mathbf{v} \in \mathbf{sOld}.o.\mathit{Elements} \cup \mathbf{sOld}.o'.\mathit{Elements}, \mathit{Elements}, \mathbf{sOld}.o.\mathit{Elements} \cup \mathbf{sOld}.o'.\mathit{Elements}\rangle$

- the list of frame rules of `addSequenceRelation` is

  - $\langle \mathbf{v} \neq \mathbf{sOld}.o.\texttt{last}, \texttt{next}\rangle$
  - $\langle \mathbf{v} \neq \mathbf{sOld}.o'.\texttt{first}, \texttt{prev}\rangle$
  - $\langle \mathbf{v} \notin \mathbf{sOld}.o.\mathit{Elements} \cup \mathbf{sOld}.o'.\mathit{Elements}, \texttt{first}\rangle$
  - $\langle \mathbf{v} \notin \mathbf{sOld}.o.\mathit{Elements} \cup \mathbf{sOld}.o'.\mathit{Elements}, \texttt{last}\rangle$
  - $\langle \mathbf{v} \notin \mathbf{sOld}.o.\mathit{Elements}, \texttt{Next}^*\rangle$
  - $\langle \mathbf{v} \notin \mathbf{sOld}.o'.\mathit{Elements}, \texttt{Prev}^*\rangle$
  - $\langle \mathbf{v} \notin \mathbf{sOld}.o.\mathit{Elements} \cup \mathbf{sOld}.o'.\mathit{Elements}, *\rangle$

We provide the formal definition of the **SSOR** for the basic operations in **Definition 50**.

Let us now consider an example of the logical characterization of the **SSOR** of a basic operation. The logical characterization of `addSequenceRelation` is $[\![\mathbf{SSOR}_{\texttt{addSequenceRelation}}]\!] = \langle \mathbf{pre}_{\texttt{addSequenceRelation}}, \mathbf{post}_{\texttt{addSequenceRelation}}\rangle$, where $\mathbf{pre}_{\texttt{addSequenceRelation}}$ is equal to $\mathbf{sOld}.o.\texttt{instID} \neq \mathbf{sOld}.o'.\texttt{instID}$ and $\mathbf{post}_{\texttt{addSequenceRelation}}$ is equal to:

$(\forall \mathbf{v} : \mathbf{v} \neq \mathbf{sOld}.o.\texttt{last} \vee \mathbf{sNew}.\mathbf{v}.\texttt{next} = \mathbf{sOld}.o'.\texttt{first})\wedge$
$(\forall \mathbf{v} : \mathbf{v} \neq \mathbf{sOld}.o'.\texttt{first} \vee \mathbf{sNew}.\mathbf{v}.\texttt{prev} = \mathbf{sOld}.o.\texttt{last})\wedge$
$(\forall \mathbf{v} : \mathbf{v} \notin \mathbf{sOld}.o.\mathit{Elements} \cup \mathbf{sOld}.o'.\mathit{Elements} \vee \mathbf{sNew}.\mathbf{v}.\texttt{first} =$
$\mathbf{sOld}.o.\texttt{first})\wedge$
$(\forall \mathbf{v} : \mathbf{v} \notin \mathbf{sOld}.o.\mathit{Elements} \cup \mathbf{sOld}.o'.\mathit{Elements} \vee \mathbf{sNew}.\mathbf{v}.\texttt{last} =$
$\mathbf{sOld}.o'.\texttt{last})\wedge$
$(\forall \mathbf{v} : \mathbf{v} \notin \mathbf{sOld}.o.\mathit{Elements} \vee \mathbf{sNew}.\mathbf{v}.\texttt{Next}^* =$
$\mathbf{sOld}.\mathbf{v}.\texttt{Next}^* \cup \mathbf{sOld}.o'.\mathit{Elements})\wedge$
$(\forall \mathbf{v} : \mathbf{v} \notin \mathbf{sOld}.o'.\mathit{Elements} \vee \mathbf{sNew}.\mathbf{v}.\texttt{Prev}^* =$
$\mathbf{sOld}.\mathbf{v}.\texttt{Prev}^* \cup \mathbf{sOld}.o.\mathit{Elements})\wedge$
$(\forall \mathbf{v} : \mathbf{v} \notin \mathbf{sOld}.o.\mathit{Elements} \cup \mathbf{sOld}.o'.\mathit{Elements} \vee \mathbf{sNew}.\mathbf{v}.\mathit{Elements} =$
$\mathbf{sOld}.o.\mathit{Elements} \cup \mathbf{sOld}.o'.\mathit{Elements})\wedge$
$(\forall \mathbf{v} : \mathbf{v} = \mathbf{sOld}.o.\texttt{last} \vee \mathbf{sNew}.\mathbf{v}.\texttt{next} = \mathbf{sOld}.\mathbf{v}.\texttt{next})\wedge$
$(\forall \mathbf{v} : \mathbf{v} = \mathbf{sOld}.o'.\texttt{first} \vee \mathbf{sNew}.\mathbf{v}.\texttt{prev} = \mathbf{sOld}.\mathbf{v}.\texttt{prev})\wedge$
$(\forall \mathbf{v} : \mathbf{v} \in \mathbf{sOld}.o.\mathit{Elements} \cup \mathbf{sOld}.o'.\mathit{Elements} \vee \mathbf{sNew}.\mathbf{v}.\texttt{first} =$
$\mathbf{sOld}.\mathbf{v}.\texttt{first})\wedge$
$(\forall \mathbf{v} : \mathbf{v} \in \mathbf{sOld}.o.\mathit{Elements} \cup \mathbf{sOld}.o'.\mathit{Elements} \vee \mathbf{sNew}.\mathbf{v}.\texttt{last} =$
$\mathbf{sOld}.\mathbf{v}.\texttt{last})\wedge$
$(\forall \mathbf{v} : \mathbf{v} \in \mathbf{sOld}.o.\mathit{Elements} \vee \mathbf{sNew}.\mathbf{v}.\texttt{Next}^* = \mathbf{sOld}.\mathbf{v}.\texttt{Next}^*)\wedge$
$(\forall \mathbf{v} : \mathbf{v} \in \mathbf{sOld}.o'.\mathit{Elements} \vee \mathbf{sNew}.\mathbf{v}.\texttt{Prev}^* = \mathbf{sOld}.\mathbf{v}.\texttt{Prev}^*)\wedge$
$(\forall \mathbf{v}, it : it \notin \mathbf{Items} \vee \mathbf{v} \in \mathbf{sOld}.o.\mathit{Elements} \cup \mathbf{sOld}.o'.\mathit{Elements} \vee$
$\mathbf{sNew}.\mathbf{v}.it = \mathbf{sOld}.\mathbf{v}.it)$

In the above example **Items** denotes the set of all stereotype items.

We use pre and post conditions of an operation to construct an axiom which describes its properties. We call this axiom the characteristic axiom of an **SSOR**.

Definition 41 (Characteristic axiom of an **SSOR**)**.** *We construct a characteristic axiom of a stereotype operation* $\boldsymbol{op}(v_1 : T_1, \ldots, v_n : T_n)$ *in the following way:* $\forall v_1, \ldots, v_n, \boldsymbol{sOld} : \exists \boldsymbol{sNew} : \boldsymbol{SysInv}[\boldsymbol{sOld}] \wedge \boldsymbol{pre_{op}} \Rightarrow \boldsymbol{post_{op}}$. *We denote this axiom as* $\boldsymbol{Ax_{op}}$.

As soon as we get the characteristic axiom from an **SSOR** we can use it as a specification. Nevertheless we would like to guarantee some extra properties of a generated specification, notably: consistency, preservation of the system invariant, and determinism.

- We call an **SSOR** consistent if its characteristic axiom is consistent. Usage of an inconsistent axiom during program verification can lead to unsoundness of the verification.

- During a verification process we would like to rely on the system invariant. That is why we require its preservation by any stereotype operation.

- There are the following obvious questions about determinism:
  - What does it mean?
  - Why do we need it?

  Let us address these questions one by one.

  - By determinism we mean that for any fixed values of input parameters and **sOld** there is not more than one value of **sNew** which satisfies the characteristic axiom. In other words a deterministic operation has to precisely describe the result of the operation execution: which stereotype items are changed and in which way. If the operation doesn't satisfy this property then there is an input such that the execution of the operation can result in one of several outcomes in a nondeterministic way. Another name for this property is absence of underspecification, but since we don't really have an implementation which could be underspecified we prefer the former name.
  - Dealing with deterministic operations is much simpler than with non-deterministic. More specifically we can generate an **SSOR** for a deterministic operation without existential quantifiers. In such a way we can restrict the language of **SSOR** to the universally quantified fragment of **FOL**. This language restriction is crucial for achieving provability of the generated specifications by **SMT** theorem provers.

    Another significant advantage of deterministic operations is that we can generate an **SSOR** for them without using intermediate stereotype slices. The idea behind it is the following one. If there is a sequential composition of two deterministic operations then we can precisely compute the intermediate slice and eliminate it. We discuss this in more detail in **Section 6.4.5**. The absence of intermediate stereotype slices makes the generated **SSOR** much simpler and therefore facilitates their automatic verification significantly.

Let us formalize the properties of an **SSOR** mentioned above.

Definition 42 (Consistency of an **SSOR**). *We call an $\boldsymbol{SSOR_{op}}$ consistent if and only if:*

- *for each disjoint i and j from $[1..n]$ if $it_i^t = it_j^t$ then*
  $\forall v_1, \ldots, v_n, \boldsymbol{sOld}, \boldsymbol{sNew} : \boldsymbol{SysInv}[\boldsymbol{sOld}] \wedge \boldsymbol{pre_{op}} \Rightarrow S_i^t \sharp S_j^t$ *holds.*
- *for each i from $[1..n]$ and j from $[1..m]$*
  $\forall v_1, \ldots, v_n, \boldsymbol{sOld}, \boldsymbol{sNew} : \boldsymbol{SysInv}[\boldsymbol{sOld}] \wedge \boldsymbol{pre_{op}} \Rightarrow S_i^t \sharp S_j^f$ *holds*

Lemma 16. *If an* $\boldsymbol{SSOR_{op}}$ *is consistent then* $\models \boldsymbol{Ax_{op}}$.

Proof. We prove the desired property by explicit model construction. We construct **sNew** which satisfies the post-condition in three steps:

- For each object o and stereotype item "it" such that there exists a transformation rule $\langle \psi_S^-, it, val \rangle$ such that $o \in S$ we set $\mathbf{sNew}.o.it$ equal to val. Here we assume that val is constructed from total functions and always produces a value. The consistency of the **SSOR** implies the absence of interference with the other transformation and frame rules. Therefore we can be sure that no inconsistency is introduced by this assignment.

- For each object o and stereotype item "it" such that there exists a frame rule $\langle \psi_S^-, it \rangle$ such that $o \in S$ we set $\mathbf{sNew}.o.it$ equal to $\mathbf{sOld}.o.it$. The consistency of the **SSOR** implies the absence of interference between transformation and frame rules. Therefore we can be sure that there are no inconsistencies between them. On the other hand it is possible that there is another frame rule $\langle \psi_{S'}^-, it \rangle$ such that $o \in S'$. But since both of the frame rules state that $\mathbf{sNew}.o.it$ is equal to $\mathbf{sOld}.o.it$ there is no contradiction between them either.

- For each object o and stereotype item "it" such that there exists no transformation rule $\langle \psi_S^-, it, val \rangle$ or frame rule $\langle \psi_S^-, it \rangle$ such that $o \in S$ we set $\mathbf{sNew}.o.it$ equal to an arbitrary value. Since the value of $\mathbf{sNew}.o.it$ is underspecified by the stereotype operation we don't create an inconsistency by this assignment either.

□

Definition 43 (System invariant preservation by an **SSOR**)**.** *An* $\boldsymbol{SSOR_{op}}$ *preserves the system invariant if and only if*
$\forall v_1, \ldots, v_n, \boldsymbol{sOld}, \boldsymbol{sNew} : \boldsymbol{SysInv}[\boldsymbol{sOld}] \wedge \boldsymbol{pre_{op}} \wedge \boldsymbol{post_{op}} \Rightarrow \boldsymbol{SysInv}[\boldsymbol{sNew}]$.

Definition 44 (Determinism of an **SSOR**)**.** *We call an* $\boldsymbol{SSOR_{op}}$ *deterministic if and only if under assumptions* $\boldsymbol{SysInv}[\boldsymbol{sOld}]$ *and* $\boldsymbol{pre_{op}}$ *for each stereotype item "it", the value of "it" is either updated by the operation or preserved. Let us define it in a more formal way. If the updates of stereotype item "it" are described by transformation rules* $\langle S_1^t, it, val_1 \rangle, \ldots, \langle S_k^t, it, val_k \rangle$ *and by a frame rule* $\langle S_f, it \rangle$ *then the operation is deterministic if and only if for each "it" the following holds*
$\forall o, v_1, \ldots, v_n, \boldsymbol{sOld}, \boldsymbol{sNew} : \boldsymbol{SysInv}[\boldsymbol{sOld}] \wedge \boldsymbol{pre_{op}} \Rightarrow \left(\bigvee_{i=1}^{k} o \in S_i^t \right) \vee \in S_f$.
In other words the values of the stereotype item in an updated stereotype slice are always specified.

Definition 45 (Proper **SSOR**)**.** *We call an* $\boldsymbol{SSOR_{op}}$ *proper if and only if it is consistent, deterministic, and preserves the system invariant.*

The last definition combines the main properties of an **SSOR**. Starting from this point we are going to work with proper **SSOR**s only.

Let us now consider several examples which illustrate how properness of **SSOR**s can be violated. Let us assume that operation **op**(o : *ref*) affects a stereotype slice of stereotype St which contains a stereotype item "*it*".

- if the stereotype operation **op** contains both transformation rules $\langle \mathbf{v} = o, it, val_1\rangle$ and $\langle \mathbf{v} = o, it, val_2\rangle$ and $val_1 \neq val_2$ then the operation is inconsistent and therefore **op** is not proper.

- if the stereotype operation **op** contains the transformation rules $\langle \mathbf{v} = o, it, val_1\rangle$ and the frame rule $\langle \mathbf{v} = o, it\rangle$ then the operation is inconsistent and therefore **op** is not proper.

- if the stereotype operation **op** contains the transformation rules $\langle \mathbf{v} = o, it, val_1\rangle$ and does not contain a frame or a transformation rule which defines updates of the stereotype item "*it*" for other objects then the operation is nondeterministic and therefore **op** is not proper.

In some cases we have to be sure that an **SSOR** affects only specific stereotype instances. We formalize this property by the following definition.

Definition 46 (Stereotype instances affected by an **SSOR**). *We say that* $\boldsymbol{SSOR_{op}}$ *affects only stereotype instances from* $\boldsymbol{Inst}_{op}$*, where* $\boldsymbol{Inst}_{op}$ *is a set of stereotype instance IDs, if and only if:*

- $\boldsymbol{null} \notin \boldsymbol{Inst}_{op}$

- $\forall i \in \boldsymbol{Inst}_{op} : i = i.\mathtt{instID}$

- *if the list of transformation rules of the operation is* $\langle \psi^-_{S^t_1}, it^t_1, val_1\rangle, \ldots, \langle \psi^-_{S^t_n}, it^t_n, val_n\rangle$ *then for each i from* $[1..n]$ $\forall v_1, \ldots, v_n, \boldsymbol{sOld}, \boldsymbol{sNew} : \boldsymbol{SysInv}[\boldsymbol{sOld}] \wedge \boldsymbol{pre_{op}}$ $\Rightarrow S^t_i \subseteq \boldsymbol{InstEl}(\boldsymbol{Inst_{op}})$ *holds.*

Let us provide several auxiliary definitions and lemmas which characterize properties of **SSOR**.

Definition 47 (Substitution in an **SSOR**). *For an* ***SSOR*** *which is defined as:*

- *a universally quantified pre-condition* ***pre***

- *a list of transformation rules* $\langle \psi^-_{S^t_1}, it^t_1, val_1\rangle, \ldots, \langle \psi^-_{S^t_n}, it^t_n, val_n\rangle$

- *a list of frame rules* $\langle \psi^-_{S^f_1}, it^f\rangle, \ldots, \langle \psi^-_{S^f_m}, it^f_m\rangle$

We define $\boldsymbol{SSOR}[v_1 := exp_1, \ldots, v_{n_{sub}} := exp_{n_{sub}}]$ *as*

- *a universally quantified pre-condition* $\boldsymbol{pre}'$
- *a list of transformation rules* $\langle \psi^{-}_{S^t_1}{}', it^t_1, val'_1\rangle, \ldots, \langle \psi^{-}_{S^t_n}{}', it^t_n, val'_n\rangle$
- *a list of frame rules* $\langle \psi^{-}_{S^f_1}{}', it^f\rangle, \ldots, \langle \psi^{-}_{S^f_m}{}', it^f_m\rangle$

where

- $\mathbf{pre}' = \mathbf{pre}[v_1 := exp_1, \ldots, v_{n_{sub}} := exp_{n_{sub}}]$
- for each i from $[1..n]$ $\psi^{-}_{S^t_i}{}' = \psi^{-}_{S^t_i}[v_1 := exp_1, \ldots, v_{n_{sub}} := exp_{n_{sub}}]$
- for each i from $[1..n]$ $val'_i = val_i[v_1 := exp_1, \ldots, v_{n_{sub}} := exp_{n_{sub}}]$
- for each i from $[1..m]$ $\psi^{-}_{S^f_i}{}' = \psi^{-}_{S^f_i}[v_1 := exp_1, \ldots, v_{n_{sub}} := exp_{n_{sub}}]$

Lemma 17. *If* $\boldsymbol{SSOR}' = \boldsymbol{SSOR}[v_1 = exp_1, \ldots, v_n = exp_n]$ *and* $[\![\boldsymbol{SSOR}]\!] = \langle \boldsymbol{pre}, \boldsymbol{post}\rangle$ *then* $[\![\boldsymbol{SSOR}']\!] = \langle \boldsymbol{pre}', \boldsymbol{post}'\rangle$*,*
where $\boldsymbol{pre}' = \boldsymbol{pre}[v_1 = exp_1, \ldots, v_n = exp_n]$ *and*
$\boldsymbol{post}' = \boldsymbol{post}[v_1 = exp_1, \ldots, v_n = exp_n]$*.*

Proof. The proof follows immediately from **Definition 47** and **Definition 40**. □

Lemma 18. *If* $\forall v_1, \ldots, v_n : \varphi$ *holds, where* φ *is a quantifier-free formula which does not contain variable* $v'_1, \ldots, v'_{n'}$*, then* $\forall v'_1, \ldots, v'_{n'} : \varphi[v_1 := exp_1, \ldots, v_n := exp_n]$ *holds, where for each i from* $[1..n']$ exp_i *contains only variables from* $v'_1, \ldots, v'_{n'}$*.*

Proof.

$$\begin{array}{ll}
\forall v_1, \ldots, v_n : \varphi & \Rightarrow \\
\forall v'_1, \ldots, v'_{n'} : \forall v_1, \ldots, v_n : \varphi & \Rightarrow \\
\forall v'_1, \ldots, v'_{n'} : \forall v_1, \ldots, v_n : (v_1 = exp_1 \wedge \ldots \wedge v_n = exp_n \Rightarrow \varphi) & \Rightarrow \\
\forall v'_1, \ldots, v'_{n'} : \varphi[v_1 := exp_1, \ldots, v_n := exp_n] &
\end{array}$$

□

Lemma 19. *If an* **SSOR** *is proper and depends only on free variables* $v_1, \ldots, v_n$*, and* $exp_1, \ldots, exp_n$ *contains only variables from* $v'_1, \ldots, v'_{n'}$*, where* $v_1, \ldots, v_n$ *and* $v'_1, \ldots, v'_{n'}$ *are pairwise disjoint, then*
$\boldsymbol{SSOR}[v_1 = exp_1, \ldots, v_n = exp_n]$ *is also proper.*

Proof. According to **Definition 45** an **SSOR** is proper if and only if it is consistent, deterministic, and preserves system invariant. We can see that all these properties are formulated as universally quantified formulas over variables $v_1, \ldots, v_n$. By applying **Lemma 18** to these properties we get that $\mathbf{SSOR}[v_1 = exp_1, \ldots, v_n = exp_n]$ is consistent, deterministic, and preserves the system invariant. This property implies that **SSOR**$[v_1 = exp_1, \ldots, v_n = exp_n]$ is proper.

$\square$

In some cases we have to check that two **SSOR**s specify the same behavior. Below we specify a syntactical procedure which we use to check this property. Later on we prove that the proposed procedure is sufficient.

Definition 48 (**SSOR** equivalence)**.** *We call $\boldsymbol{SSOR}_1$ and $\boldsymbol{SSOR}_2$ equivalent, denoted as $\boldsymbol{SSOR}_1 \asymp \boldsymbol{SSOR}_2$, if and only if:*

- *They depend only on the same free variables $v_1, \ldots, v_n$*
- *They are both consistent and deterministic.*
- $\forall v_1, \ldots, v_n, \boldsymbol{sOld} : (\boldsymbol{SysInv}[\boldsymbol{sOld}] \wedge \boldsymbol{pre}_1) \Rightarrow \boldsymbol{pre}_2$
- $\forall v_1, \ldots, v_n, \boldsymbol{sOld} : (\boldsymbol{SysInv}[\boldsymbol{sOld}] \wedge \boldsymbol{pre}_2) \Rightarrow \boldsymbol{pre}_1$
- *for each $i_1 \in [1..n_t^1]$ and $i_2 \in [1..n_t^2]$ if $it_{i_1,1}^t = it_{i_2,2}^t$ then* $\forall v_1, \ldots, v_n, \boldsymbol{sOld}, \boldsymbol{v} : (\boldsymbol{SysInv}[\boldsymbol{sOld}] \wedge \boldsymbol{pre}_1 \wedge \varphi^-_{S_{i_1,1}^t} \wedge \varphi^-_{S_{i_2,2}^t}) \Rightarrow val_{i_1,1}^t = val_{i_2,2}^t$.
- *for each $i_1 \in [1..n_t^1]$ and $i_2 \in [1..n_f^2]$ if $it_{i_1,1}^t = it_{i_2,2}^f$ then* $\forall v_1, \ldots, v_n, \boldsymbol{sOld} : (\boldsymbol{SysInv}[\boldsymbol{sOld}] \wedge \boldsymbol{pre}_1) \Rightarrow S_{i_1,1}^t \sharp S_{i_2,2}^f$.
- *for each $i_1 \in [1..n_f^1]$ and $i_2 \in [1..n_t^2]$ if $it_{i_1,1}^f = it_{i_2,2}^t$ then* $\forall v_1, \ldots, v_n, \boldsymbol{sOld} : (\boldsymbol{SysInv}[\boldsymbol{sOld}] \wedge \boldsymbol{pre}_1) \Rightarrow S_{i_1,1}^f \sharp S_{i_2,2}^t$.

where $\boldsymbol{SSOR}_1$ has the following form:

- *pre-condition $\boldsymbol{pre}_1$.*
- *transformation rules are $\langle \psi^-_{S_{1,1}^t}, it_{1,1}^t, val_{1,1} \rangle, \ldots, \langle \psi^-_{S_{n_t^1,1}^t}, it_{n_t^1,1}, val_{n_t^1,1} \rangle$.*
- *frame rules are $\langle \psi^-_{S_{1,1}^f}, it_{1,1}^f \rangle, \ldots, \langle \psi^-_{S_{n_f^1,1}^f}, it_{n_f^1,1} \rangle$.*

and $\boldsymbol{SSOR}_2$ has the following form:

- *pre-condition $\boldsymbol{pre}_2$.*
- *transformation rules are $\langle \psi^-_{S_{1,2}^t}, it_{1,2}^t, val_{1,2} \rangle, \ldots, \langle \psi^-_{S_{n_t^2,2}^t}, it_{n_t^2,2}, val_{n_t^2,2} \rangle$.*
- *frame rules are $\langle \psi^-_{S_{1,2}^f}, it_{1,2}^f \rangle, \ldots, \langle \psi^-_{S_{n_f^2,2}^f}, it_{n_f^2,2} \rangle$.*

6.3 Specification language of specifications

Let us now consider the specification language of specifications (denoted as **SLS**) which we use to specify stereotype operations. The basic building blocks of **SLS** are basic operations. They represent primitive updates of stereotype slices. We combine them using operation composers into composite stereotype operations. There are two kinds of composite stereotype operations: non-recursive and recursive. Non-recursive operations are essentially specified by an **SLS** term. A specification of a recursive operation contains also an inductive hypothesis and a measure. In **SLS** recursive operations play the role of loops or inductive definitions. We consider them in more details later.

We begin the description of **SLS** with basic operations. Below we provide a definition of a basic operation and a description of how to construct an **SSOR** for it. We already know how to construct a pair of pre and post conditions from an **SSOR**. We use the pre and post conditions of a basic operation to define its operational semantic.

Definition 49 (Basic operation)**.** *The description of a basic operation consists of the following parts:*

- `Header:`
  $\boldsymbol{op}(v_1 : T_1, \ldots, v_n : T_n)$ *here* ***op*** *is the name of the basic operation,* $v_1, \ldots, v_n$ *are the names of the input parameters, and* $T_1, \ldots, T_n$ *are the types of the input parameters.*

- `Pre-conditions:`
  A pre-condition is a universally quantified formula which depends on the input parameters and the input stereotype slice only. We denote it as $\boldsymbol{pre_{op}}$.

- `Local variables:`
  We use local variables to abbreviate the descriptions of stereotype operations. There are local variables of two types: scalars and maps.

  – *A definition of a scalar variable has the form* $v = exp$*, where* v *is a variable name and* exp *is an expression.*

  – *A definition of a map variable has the form* $\forall o_1, \ldots, o_n : v[o_1, \ldots, o_n] = exp[o_1, \ldots, o_n]$*, where* v *is a variable name and* exp *is an expression.*

  It is assumed that there is no cyclic dependance between variable declarations. The check of this property is straightforward. We can mention *local variables at any part of the operation declaration.*

- `Input instances:`
  This part of the operation description lists all stereotype instances which can be affected by an operation execution. We call such instances participating. The description has the form $inInst_1 = exp_1^{in}; \ldots ; inInst_{n_{in}} = exp_{n_{in}}^{in}$. *Here* exp_i^{in} *for* i *in* $[1..n_{in}]$ *are expressions of type* ref *which contain IDs of the participating instances. We denote* $\{exp_1^{in}, \ldots, exp_{n_{in}}^{in}\}$ *as* $\boldsymbol{Inst}_{op}$.

- `Output instances:`
  In this section we describe how the shape of the participating instances will change after the operation execution. The description has the following form: $outInst_1 = exp_1^{out}; \ldots ; outInst_{n_{out}} = exp_{n_{out}}^{out}$. *Here* exp_i^{out} *for* i *in* $[1..n_{in}]$ *are expressions of type* Reg *which contain elements of the corresponding output instances.*

- `Transformations:`
  The last section describes how the operation transforms an input slice. It contains a list of transformation rules $\langle \psi_{S_1^t}^{-}, it_1^t, val_1 \rangle, \ldots,$ $\langle \psi_{S_{n_t}^t}^{-}, it_{n_t}, val_{n_t} \rangle$. *Here we assume that* $it_i^t \neq$ Elements *for all* i *in* $[1..n_t]$. *By this restriction we avoid possible conflicts with the specifications which we generate from input and output instances. Another restriction is that for each* i *in* $[1..n_t]$ *we assume that* $\psi_{S_i^t}^{-}$ *is quantifier free. By this restriction we enable the automatic generation of frame rules from the transformation rules. On the other hand, since we use basic operations only to express the simplest primitive transformations of stereotype slices, the basic operations are expressive enough even with this restriction. Both constraints can be checked in a simple syntactic way.*

We can eliminate any reference to a local variable in the following way:

- If a formula $\varphi[v]$ depends on a local variable $v = exp$ then it is equivalent to $\varphi[v := exp]$

- If a formula $\varphi[v[exp_1, \ldots, exp_n]]$ depends on a local variable $\forall o_1, \ldots, o_n : v[o_1, \ldots, o_n] = exp[o_1, \ldots, o_n]$ then it is equivalent to $\varphi[v := exp[exp_1, \ldots, exp_n]]$

Definition 50 (**SSOR** for a basic operation)**.** *We define an* ***SSOR*** *for a basic operation* ***op*** *in the following way:*

- *pre-condition* $\boldsymbol{pre_{op}}$
- *transformation rules are* $\langle \psi^-_{S^t_1}, it^t_1, val_1\rangle, \ldots, \langle \psi^-_{S^t_{n_t}}, it_{n_t}, val_{n_t}\rangle$ *and* $\langle exp^{out}_1, \mathrm{Elements}, exp^{out}_1\rangle, \ldots, \langle exp^{out}_{n_{out}}, \mathrm{Elements}, exp^{out}_{n_{out}}\rangle$.
- *frame rules are* $\langle \neg\psi^-_{S^t_1}, it^t_1\rangle, \ldots, \langle \neg\psi^-_{S^t_{n_t}}, it_{n_t}\rangle$ *and* $\langle \overline{exp^{out}_1 \cup \ldots \cup exp^{out}_{n_{out}}}, *\rangle$.

If the definition of a stereotype operation does not contain a transformation rule which affects a stereotype item "it" then we assume that the values of the stereotype item are preserved and add the frame rule $\langle \mathbb{U}, it\rangle$.

To initialize stereotype items of a freshly allocated object we use a special kind of basic stereotype operations: stereotype constructors.

Definition 51 (Stereotype constructor)**.** *A stereotype constructor consists of:*

- `Header:`
  `constructor` $\boldsymbol{op}(obj : \mathrm{ref!}, v_1 : T_1, \ldots, v_n : T_n)$ *here* $\boldsymbol{op}$ *is the name of the constructor,* $obj, v_1, \ldots, v_n$ *are the names of the input parameters, and* $\mathrm{ref!}, T_1, \ldots, T_n$ *are the types of input parameters. The first input parameter* obj *has a special meaning. It is a reference to a freshly allocated object.*

- `Body:`
  The body contains a list of transformation rules of the form $\langle obj, it_1, exp_1\rangle, \ldots, \langle obj, it_n, exp_n\rangle$. *We assume that the transformation rules satisfy the following syntactical constraints:*

  1. *For each stereotype item there is exactly only transformation rule.*
  2. *Each transformation rule has the form* $\langle obj, it^t, val\rangle$.
  3. *For each transformation rule* $\langle obj, it^t, val\rangle$ *where* it^t *is a stereotype item of type* Reg, val *is* $\{obj\}$ *or* $\varnothing$.
  4. *For each transformation rule* $\langle obj, it^t, val\rangle$ *where* it^t *is a stereotype item of type* ref, val *is* obj *or* $\boldsymbol{null}$.
  5. *for each* i *in* $[1..n]$, val_i *does not depend on the old values of stereotype items of* obj.

Definition 52 (**SSOR** for a constructor)**.** *We define an* ***SSOR*** *for a constructor* $\boldsymbol{op}$ *as:*

- *pre-condition* T

- *transformation rules are* $\langle obj, it_1, exp_1\rangle, \ldots, \langle obj, it_n, exp_n\rangle$ *and* $\langle obj, \text{Elements}, \{obj\}\rangle$.
- *frame rule is* $\langle \overline{\{obj\}}, *\rangle$.

$$\begin{array}{lcl}
\mathbf{t} & ::= & \mathbf{SSOR} \\
 & | & \mathbf{op}(exp_1, \ldots, \text{exp}_n) \\
 & | & \texttt{if } (\varphi)\ \mathbf{t}\ \texttt{else}\ \mathbf{t} \\
 & | & \mathbf{t}; \mathbf{t} \\
 & | & \overset{f_{\mathbf{Inst}}}{\underset{i \in Ind}{\|}} \mathbf{op}(exp_1[i], \ldots, \text{exp}_n[i]) \\
 & | & \texttt{skip}
\end{array}$$

Figure 6.1: **SLS** syntax.

The syntax and the operational semantics of **SLS** are presented on Figure 6.1 and Figure 6.2, respectively. **SLS** consists of the following expressions: **SSOR** execution, operation call, conditional statement, sequential composition, parallel composition, and the skip operation. Since we use them to compose a more complicated stereotype operation from the simpler stereotype operations we also call them operation composers.

A transition in the operational semantics has the following form $\Sigma \models \mathbf{s} \xrightarrow{\mathbf{t}} \mathbf{s}'$. The judgment states that under environment Σ term $\mathbf{t}$ transforms an initial state $\mathbf{s}$ into a final state $\mathbf{s}'$. Here an environment Σ defines values of variables and has the following structure $\Sigma = \{v_1 \mapsto val_1, \ldots, v_n \mapsto val_n\}$, where v_i is a variable name and val_i is a variable value. The state is the state of the stereotype slice affected by the operation execution.

We denote the interpretation of a variable v under environment Σ as $\Sigma[\![v]\!]$, the interpretation of a formula φ under environment Σ as $\Sigma \models \varphi$, and the interpretation of an expression exp under environment Σ as $[\![\Sigma]\!]_{exp}$. Since all of them are standard we omit their definitions.

Formulas and expressions in **SLS** terms depend on a slice. To make our reasoning simpler we assume that this slice is denoted as **sOld**.

A basic building block of **SLS** is an **SSOR**. We describe an operational aspect of the behavior of an **SSOR** in a logical way. An execution of an **SSOR** transforms an input stereotype slice into an output stereotype slice if and only if the system invariant, pre-condition, and post-condition hold for these slices.

The operational semantics of a basic and a composite operation call differ. A call of a basic operation is reduced to an execution of the **SSOR** that corresponds to the body of the operation. On the other hand an execution of a composite operation is equivalent to an execution of its body. There is an extra check for a recursive call of a recursive operation. In this case we

$$\text{OS-SSOR}\ \frac{\begin{array}{c}\Sigma' = \Sigma \cup \{\mathbf{sOld} \mapsto \mathbf{s}, \mathbf{sNew} \mapsto \mathbf{s}'\} \\ [\![\mathbf{SSOR}]\!] = \langle \mathbf{pre}, \mathbf{post} \rangle \\ \Sigma' \models \mathbf{SysInv}[\mathbf{sOld}] \wedge \mathbf{pre} \wedge \mathbf{post}\end{array}}{\Sigma \models \mathbf{s} \overset{\mathbf{SSOR}}{\to} \mathbf{s}'}$$

$$\text{OS-Call-Basic}\ \frac{\begin{array}{c}\mathbf{op}(v_1 : T_1, \ldots, v_n : T_n) \text{ is a basic stereotype operation} \\ \text{for each } i \text{ from } [1..n]\ exp_i \text{ is a well-formed expression of the type } T_i \\ \Sigma' = \Sigma \cup \{\mathbf{sOld} \mapsto \mathbf{s}\} \\ \Sigma'' = \{v_1 \mapsto [\![exp_1]\!]_{\Sigma'}, \ldots, v_n \mapsto [\![exp_n]\!]_{\Sigma'}\} \\ \Sigma'' \models \mathbf{s} \overset{\mathbf{SSOR_{op}}}{\to} \mathbf{s}'\end{array}}{\Sigma \models \mathbf{s} \overset{\mathbf{op}(exp_1, \ldots, exp_n)}{\to} \mathbf{s}'}$$

$$\text{OS-Call-Comp}\ \frac{\begin{array}{c}\mathbf{op}(v_1 : T_1, \ldots, v_n : T_n) \text{ is a composite stereotype operation} \\ \text{for each } i \text{ from } [1..n]\ exp_i \text{ is a well-formed expression of the type } T_i \\ \mathbf{t_{op}} \text{ is the body of } \mathbf{op} \\ \Sigma' = \Sigma \cup \{\mathbf{sOld} \mapsto \mathbf{s}\} \\ \Sigma'' = \{v_1 \mapsto [\![exp_1]\!]_{\Sigma'}, \ldots, v_n \mapsto [\![exp_n]\!]_{\Sigma'}\} \\ \text{if the call is recursive and } v_{\mathbf{mes}} \text{ is a measure variable} \\ \text{then } \Sigma''[\![v_{\mathbf{mes}}]\!] < \Sigma[\![v_{\mathbf{mes}}]\!] \\ \Sigma'' \models \mathbf{s} \overset{\mathbf{t_{op}}}{\to} \mathbf{s}'\end{array}}{\Sigma \models \mathbf{s} \overset{\mathbf{op}(exp_1, \ldots, exp_n)}{\to} \mathbf{s}'}$$

$$\text{OS-if}\ \frac{\begin{array}{c}\Sigma' = \Sigma \cup \{\mathbf{sOld} \mapsto \mathbf{s}\} \\ (\Sigma' \models \quad \varphi) \text{ implies } \Sigma \models \mathbf{s} \overset{\mathbf{t_1}}{\to} \mathbf{s}' \\ (\Sigma' \models \neg\varphi) \text{ implies } \Sigma \models \mathbf{s} \overset{\mathbf{t_2}}{\to} \mathbf{s}'\end{array}}{\Sigma \models \mathbf{s} \overset{\mathtt{if}\ (\varphi)\ \mathbf{t_1}\ \mathtt{else}\ \ \mathbf{t_2}}{\to} \mathbf{s}'} \qquad \text{OS-seq}\ \frac{\begin{array}{c}\Sigma \models \mathbf{s} \overset{\mathbf{t_1}}{\to} \mathbf{s}'' \\ \Sigma \models \mathbf{s}'' \overset{\mathbf{t_2}}{\to} \mathbf{s}'\end{array}}{\Sigma \models \mathbf{s} \overset{\mathbf{t_1;t_2}}{\to} \mathbf{s}'}$$

$$\text{OS-par}\ \frac{\begin{array}{c}\text{for each } i \in Ind\ \mathbf{op}(exp_1[i], \ldots, exp_n[i]) \text{ effects only} \\ \text{stereotype instances from } \mathbf{Inst_{op}}[i] \\ \Sigma' = \Sigma \cup \{\mathbf{sOld} \mapsto \mathbf{s}\} \\ \Sigma'' = \Sigma \cup \{\mathbf{sOld} \mapsto \mathbf{s}, \mathbf{sNew} \mapsto \mathbf{s}'\} \\ \text{for each disjoint } i, j \in Ind\ \Sigma' \models \mathbf{Inst_{op}}[i] \sharp \mathbf{Inst_{op}}[j] \\ \mathbf{Inst} = \bigcup_{i \in Ind}^{f_{\mathbf{Inst}}} \mathbf{Inst_{op}}[i] \\ \Sigma' \models \bigcup_{i \in Ind} \mathbf{Inst_{op}}[i] = \mathbf{Inst} \\ \Sigma'' \models [\![\langle \overline{\mathbf{InstEl}(\mathbf{Inst})}, * \rangle]\!] \\ \text{for each } i \in Ind\ \Sigma \models \mathbf{s} \overset{\mathbf{op}(exp_1[i], \ldots, exp_n[i]) \blacktriangleleft \mathbf{Inst_{op}}[i]}{\to} \mathbf{s}'\end{array}}{\Sigma \models \mathbf{s} \overset{\overset{f_{\mathbf{Inst}}}{\underset{i \in Ind}{\|}} \mathbf{op}(exp_1[i], \ldots, exp_n[i])}{\to} \mathbf{s}'}$$

$$\text{OS-rest}\ \frac{\begin{array}{c}\Sigma \models \mathbf{s} \overset{\mathbf{t}}{\to} \mathbf{s}'' \\ \Sigma' = \Sigma \cup \{\mathbf{sOld} \mapsto \mathbf{s}'', \mathbf{sNew} \mapsto \mathbf{s}'\} \\ \Sigma' \models [\![\langle \mathbf{InstEl}(Inst), * \rangle]\!]\end{array}}{\Sigma \models \mathbf{s} \overset{\mathbf{t} \blacktriangleleft \mathbf{Inst}}{\to} \mathbf{s}'} \qquad \text{OS-skip}\ \frac{}{\Sigma \models \mathbf{s} \overset{\mathsf{skip}}{\to} \mathbf{s}}$$

Figure 6.2: **SLS** operational semantics.

require the value of the measure variable to decrease. By this we guarantee termination of a recursive call and prevent unsoundness. A measure variable is a part of the definition of a composite recursive operation which we discuss later.

The semantics of both conditional statement and sequential composition are standard. If the condition of a conditional statement holds, then the first term is executed, otherwise the second. A sequential composition transforms an input term into an output term if and only if there exists an intermediate stereotype slice such that the first term transforms the input slice into the intermediate slice and the second term transforms the intermediate slice into the output slice.

The next operation composer is the parallel composition. The main idea behind it is that if there are several operations whose effects are disjoint then the order of their execution doesn't matter and we can execute them simultaneously. We guarantee disjointness of the operations' effects by requiring that each operation affects only stereotype instances from $\mathbf{Inst}_{\mathbf{op}}[i]$ and by requiring disjointness of $\mathbf{Inst}_{\mathbf{op}}[i]$. Since information about the affected stereotype instances is unknown for an arbitrary stereotype operation we allow parallel composition only for basic and recursive stereotype operations. For both of them, the affected stereotype instances are a part of user-provided specifications.

To define the semantics of the parallel composition we introduce an auxiliary stereotype composer: the restriction of an operation to a set of stereotype instances. We denote it as $\mathbf{t} \blacktriangleleft \mathbf{Inst}$. For all objects which belong to the stereotype instances from **Inst**, the restricted operation has the same effect as the non-restricted operation. The values of all objects outside the stereotype instances from **Inst** are unspecified. We achieve this behavioral nondeterminism by using existential quantification over the stereotype slice. $\mathbf{s}''$ is the output of the unrestricted operation. **sNew** equals $\mathbf{s}''$ for all objects from stereotype instances from **Inst** and is not specified for the rest of the objects.

Using operation restriction we can specify parallel composition in the following way. $f_{\mathbf{Inst}}$ is an auxiliary parameter of the parallel composition. We use $f_{\mathbf{Inst}}$ as a witness function to define set $\mathbf{Inst} = \bigcup_{i \in Ind}^{f_{\mathbf{Inst}}} \mathbf{Inst}_{\mathbf{op}}[i]$. **Inst** is the set of identifiers of stereotype instances which are affected by the parallel composition. Values of all objects outside the instances from **Inst** are preserved. An updated value of an object from $\mathbf{Inst}_{\mathbf{op}}[i]$ is equal to the parallel execution of the restricted versions of the operations $\mathbf{op}(exp_1[i], \ldots, exp_n[i])$. We simulate parallel execution by using a universal quantifier. Since each of the operations $\mathbf{op}(exp_1[i], \ldots, exp_n[i])$ states that the values of all objects outside the stereotype instance $\mathbf{Inst}_{\mathbf{op}}[i]$ are preserved, we use the restricted version of operations to avoid inconsistency. The restricted versions of the operations $\mathbf{op}(exp_1[i], \ldots, exp_n[i])$ leave values of the objects outside the stereotype instances $\mathbf{Inst_{op}}[i]$ unspecified and therefore inconsistency is avoided.

The last operation is `skip`. It is an identical transformation of an input slice. Mostly it is used when we want to specify that an operation doesn't change a stereotype slice.

Let us now consider composite stereotype operations. As we mentioned above, there are two kinds of composite stereotype operations; non-recursive and recursive. Let us begin with the first one.

Definition 53 (Non-recursive composite stereotype operation)**.** *A description of a non-recursive composite stereotype operation consists of the following parts:*

- *`Header:`*
  $\boldsymbol{op}(v_1 : T_1, \ldots, v_n : T_n)$ *here* ***op*** *is a name of an operation,* $v_1, \ldots, v_n$ *are the names of the input parameters, and* $T_1, \ldots, T_n$ *are the types of the input parameters.*

- *`Body:`*
  is an ***SLS*** *term* $\boldsymbol{t}$ *which possibly depends on the variables* $v_1, \ldots, v_n$ *and does not contain recursive calls. We use a pair of curly brackets to identify the beginning and the end of a body.*

Definition 54 (Recursive composite stereotype operation)**.** *A description of a recursive composite stereotype operation consists of the following parts:*

- *`Header:` The same as the corresponding part of the description of a basic stereotype operation.*

- *`Pre-conditions:` The same as the corresponding part of the description of a basic stereotype operation.*

- *`Local variables:` The same as corresponding part of the description of a basic stereotype operation.*

- *`Input instances:`*
  Similar to the corresponding part of the description of a basic stereotype operation but instead of a fixed number of input stereotype instances we have a set of input instances. Because of this extension, recursive operations are able to deal with an unbounded number of input instances.

- `Transformations:`
  Similar to the corresponding part of a the description of a basic stereotype operation but with two differences. First we allow the $\psi^-_{S^t}$ of a transformation rule $\langle \psi^-_{S^t}, it, val \rangle$ to contain an arbitrary number of existential quantifiers. By this we extend the expressive power of recursive operations. Second, we require an explicit description of updates of the Elements *stereotype item. We have to describe them explicitly because we drop the "output instances" section. We can drop it because we don't need to check preservation of* ***SysInvEl****. Later on we will prove that the way how we construct recursive operations guarantees the preservation of* ***SysInvEl****.*

- `Frame:`
  Since we allow $\psi^-_{S^t}$ to contain an arbitrary number of existential quantifiers, which means that $\neg\psi^-_{S^t}$ is potentially universally quantified, we can't infer frame properties anymore. Therefore we have to describe them in the following explicit form $\langle \psi^-_{S^f_1}, it^f_1 \rangle, \ldots, \langle \neg\psi^-_{S^f_{n_f}}, it_{n_f} \rangle$.

- `Measure:`
  Here we provide the name of an input parameter which is used as a measure and guarantees termination of recursive calls. We denote it as $v_{\boldsymbol{mes}}$. We also denote an expression which represents an actual value of $v_{\boldsymbol{mes}}$ as $\exp_{\boldsymbol{mes}}$. The type of the variable has to be equipped with a well-founded order. For example we use strictly less order for positive numbers and strict inclusion order for sets.

- `Body:`
  The body is similar to the corresponding part of the description of a non-recursive composite stereotype operation. The only difference is that ***t*** *possibly contains recursive calls of the defining stereotype operation.*

A definition of a recursive operation consists of two equivalent descriptions: operational and logical. An operational description is represented by the operation body which we use to define the operational semantics of a recursive operation call. A logical description is represented by a precondition, transformation rules, and frame rules. We use the logical description to construct an **SSOR** for the recursive operation. On the other hand we use the operational description and measure to prove that the constructed **SSOR** is proper.

If the type of the measure of a recursive operation is a set type, then in order to check the preservation of the measure by a recursive call we have to check a strict set inclusion. As we mentioned above to avoid existential quantifiers in a property's check we have to use strict subset inclusion with a witness element. With this motivation in mind, we annotate this recursive call with a hint which witness element to use. To do that we use the following syntax $\mathbf{op}(\ldots, exp, \ldots)$ `measure` $v \overset{exp'}{\subseteq} exp$, where v is the measure variable of the recursive operation **op**, exp of type *Reg* is an actual value of the measure of the recursive call, and exp' of type *ref* is a witness value which guarantees that the measure decreases.

Before we move on to the consideration of the **SLS**to-**SSOR** translation we have to introduce an equivalence relation induced by the operational semantics on sets of **SSOR**s and prove some of its properties.

Definition 55 (Equivalence relation on the **SLS** terms for a fixed Σ)**.** *Two **SLS** terms $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$ which depend on the same free variables $v_1, \ldots, v_n$, are equivalent for an environment $\Sigma = \{v_1 \mapsto val_1, \ldots, v_n \mapsto val_n\}$ if and only if for any stereotype slices s and s' the following holds: $\Sigma \models s \xrightarrow{\boldsymbol{t}_1} s'$ if and only if $\Sigma \models s \xrightarrow{\boldsymbol{t}_2} s'$. We denote equivalence of terms $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$ for an environment Σ as $\boldsymbol{t}_1 \overset{\Sigma}{=} \boldsymbol{t}_2$.*

Lemma 20. *For each fixed environment Σ $\overset{\Sigma}{=}$ is an equivalence relation.*

Proof.
- reflexivity: for each $\mathbf{t}$, s, and s', $\Sigma \models s \xrightarrow{\mathbf{t}} s'$ if and only if $\Sigma \models s \xrightarrow{\mathbf{t}} s'$ which implies $\mathbf{t} \overset{\Sigma}{=} \mathbf{t}$.

- symmetric: for each $\mathbf{t}$ and $\mathbf{t}'$ if $\mathbf{t} \overset{\Sigma}{=} \mathbf{t}'$ then for each s and s' $\Sigma \models s \xrightarrow{\mathbf{t}} s'$ if and only if $\Sigma \models s \xrightarrow{\mathbf{t}'} s'$. It implies $\Sigma \models s \xrightarrow{\mathbf{t}'} s'$ if and only if Σ $\Sigma \models s \xrightarrow{\mathbf{t}} s'$ which is equivalent to $\mathbf{t}' \overset{\Sigma}{=} \mathbf{t}$.

- transitivity: for each $\mathbf{t}$, $\mathbf{t}'$, and $\mathbf{t}''$ if $\mathbf{t} \overset{\Sigma}{=} \mathbf{t}'$ and $\mathbf{t}' \overset{\Sigma}{=} \mathbf{t}''$ then for each s and s' $\Sigma \models s \xrightarrow{\mathbf{t}} s'$ if and only if $\Sigma \models s \xrightarrow{\mathbf{t}'} s'$ and $\Sigma \models s \xrightarrow{\mathbf{t}'} s'$ if and only if $\Sigma \models s \xrightarrow{\mathbf{t}''} s'$. It implies $\Sigma \models s \xrightarrow{\mathbf{t}} s'$ if and only if $\Sigma \models s \xrightarrow{\mathbf{t}''} s'$ which is equivalent to $\mathbf{t} \overset{\Sigma}{=} \mathbf{t}''$.

□

Lemma 21. *For each $\boldsymbol{t}$, $\boldsymbol{t}_1$, $\boldsymbol{t}_2$, and Σ, $\boldsymbol{t}_1 \overset{\Sigma}{=} \boldsymbol{t}_2$* (1) *implies $\boldsymbol{t}[\boldsymbol{t}_1 := \boldsymbol{t}_2] \overset{\Sigma}{=} \boldsymbol{t}$.*

Proof. The desired property can be proven by structural induction on the term $\mathbf{t}$.

- **Induction base:**
  - $\mathbf{t}$ is equal to $\mathbf{t}_1$ (2). (2) implies $\mathbf{t}[\mathbf{t}_1 := \mathbf{t}_2] = \mathbf{t}_1[\mathbf{t}_1 := \mathbf{t}_2] = \mathbf{t}_2$ (3). (1) $\wedge$ (3) $\Rightarrow \mathbf{t}[\mathbf{t}_1 := \mathbf{t}_2] \overset{\Sigma}{=} \mathbf{t}_1 = \mathbf{t}$.

  - $\mathbf{t}$ differs from $\mathbf{t}_1$ (4) and $\mathbf{t}$ is **SSOR**, $\mathbf{op}(exp_1, \ldots, exp_n)$, or `skip` (5). (4) $\wedge$ (5) $\Rightarrow \mathbf{t}[\mathbf{t}_1 := \mathbf{t}_2] = \mathbf{t} \overset{\Sigma}{=} \mathbf{t}$.

- **Induction step:** For each $\mathbf{s}$ and $\mathbf{s}'$ we have to prove that $\Sigma \models s \overset{\mathbf{t}[\mathbf{t}_1:=\mathbf{t}_2]}{\rightarrow} s'$ holds if and only if $\Sigma \models s \overset{\mathbf{t}}{\rightarrow} s'$.

  - $\mathbf{t}$ is equal to `if` (φ) $\mathbf{t}'$ `else` $\mathbf{t}''$ and differs from $\mathbf{t}_1$. Let us first consider the case when $\Sigma' \models \varphi$ holds (6), where $\Sigma' = \Sigma \cup \{\mathbf{sOld} \mapsto \mathbf{s}\}$. $\Sigma \models s \overset{\mathbf{t}[\mathbf{t}_1:=\mathbf{t}_2]}{\rightarrow} s'$ is equivalent to $\Sigma \models s \overset{\texttt{if}\ (\varphi)\ \mathbf{t}'[\mathbf{t}_1:=\mathbf{t}_2]\ \texttt{else}\ \ \mathbf{t}''[\mathbf{t}_1:=\mathbf{t}_2]}{\rightarrow} s'$. According to (6) it is equivalent to $\Sigma \models s \overset{\mathbf{t}'[\mathbf{t}_1:=\mathbf{t}_2]}{\rightarrow} s'$. According to the induction hypothesis it is equivalent to $\Sigma \models s \overset{\mathbf{t}'}{\rightarrow} s'$. According to (6) it is equivalent to $\Sigma \models s \overset{\mathbf{t}}{\rightarrow} s'$. The proof for the case $\Sigma' \models \neg\varphi$ can be constructed in a similar way.

  - $\mathbf{t}$ is equal to $\mathbf{t}'; \mathbf{t}''$ and differs from $\mathbf{t}_1$. $\Sigma \models s \overset{\mathbf{t}[\mathbf{t}_1:=\mathbf{t}_2]}{\rightarrow} s'$ is equivalent to $\Sigma \models s \overset{\mathbf{t}'[\mathbf{t}_1:=\mathbf{t}_2];\mathbf{t}''[\mathbf{t}_1:=\mathbf{t}_2]}{\rightarrow} s'$. According to the operational semantic it is equivalent to $\exists\ s'' : \Sigma \models s \overset{\mathbf{t}'[\mathbf{t}_1:=\mathbf{t}_2]}{\rightarrow} s'' \wedge \Sigma \models s'' \overset{\mathbf{t}''[\mathbf{t}_1:=\mathbf{t}_2]}{\rightarrow} s'$. According to the induction hypothesis it is equivalent to $\exists\ s'' : \Sigma \models s \overset{\mathbf{t}'}{\rightarrow} s'' \wedge \Sigma \models s'' \overset{\mathbf{t}''}{\rightarrow} s'$. According to the operational semantic it is equivalent to $\Sigma \models s \overset{\mathbf{t}';\mathbf{t}''}{\rightarrow} s'$. It is equivalent to $\Sigma \models s \overset{\mathbf{t}}{\rightarrow} s'$.

  - $\mathbf{t}$ is equal to $\mathbf{t}' \blacktriangleleft \mathbf{Inst}$ and differs from $\mathbf{t}_1$. $\Sigma \models s \overset{\mathbf{t}[\mathbf{t}_1:=\mathbf{t}_2]}{\rightarrow} s'$ is equivalent to $\Sigma \models s \overset{\mathbf{t}'[\mathbf{t}_1:=\mathbf{t}_2]\blacktriangleleft\mathbf{Inst}}{\rightarrow} s'$. According to the operational semantic it is equivalent to $\exists\ \mathbf{s}'' : \Sigma \models \mathbf{s} \overset{\mathbf{t}'[\mathbf{t}_1:=\mathbf{t}_2]}{\rightarrow} \mathbf{s}'' \wedge \Sigma' \models [\![\langle \mathbf{InstEl}(Inst), * \rangle]\!]$, where $\Sigma' = \Sigma \cup \{\mathbf{sOld} \mapsto \mathbf{s}'', \mathbf{sNew} \mapsto \mathbf{s}'\}$. According to the induction hypothesis it is equivalent to $\exists\ \mathbf{s}'' : \Sigma \models \mathbf{s} \overset{\mathbf{t}'}{\rightarrow} \mathbf{s}'' \wedge \Sigma' \models [\![\langle \mathbf{InstEl}(Inst), * \rangle]\!]$. According to the operational semantic it is equivalent to $\Sigma \models s \overset{\mathbf{t}'\blacktriangleleft\mathbf{Inst}}{\rightarrow} s'$. It is equivalent to $\Sigma \models s \overset{\mathbf{t}}{\rightarrow} s'$.

  - $\mathbf{t}$ is equal to $\overset{f_{\mathbf{Inst}}}{\underset{i \in Ind}{\|}} \mathbf{op}(exp_1[i], \ldots, exp_n[i])$ and differs from $\mathbf{t}_1$. $\Sigma \models s \overset{\mathbf{t}[\mathbf{t}_1:=\mathbf{t}_2]}{\rightarrow} s'$ is equivalent to $\Sigma \models s \overset{\overset{f_{\mathbf{Inst}}}{\underset{i \in Ind}{\|}} (\mathbf{op}(exp_1[i],\ldots,exp_n[i])[\mathbf{t}_1:=\mathbf{t}_2])}{\rightarrow} s'$. According to the operational semantic it is equivalent to $(\forall\ i \in Ind : \Sigma \models \mathbf{s} \overset{\mathbf{op}(exp_1[i],\ldots,exp_n[i])[\mathbf{t}_1:=\mathbf{t}_2]\blacktriangleleft\mathbf{Inst}_{\mathbf{op}}[i]}{\rightarrow} \mathbf{s}') \wedge \varphi$, where
    $\varphi \Leftrightarrow (\forall i, j \in Ind : i \neq j \Rightarrow \Sigma' \models \mathbf{Inst}_{\mathbf{op}}[i] \sharp \mathbf{Inst}_{\mathbf{op}}[j]) \wedge$
    $(\mathbf{Inst} = \bigcup\limits_{i \in Ind}^{f_{\mathbf{Inst}}} \mathbf{Inst}_{\mathbf{op}}[i]) \wedge (\Sigma' \models \bigcup\limits_{i \in Ind} \mathbf{Inst}_{\mathbf{op}}[i] = \mathbf{Inst}) \wedge$
    $(\Sigma'' \models [\![\langle \overline{\mathbf{InstEl}(\mathbf{Inst})}, * \rangle]\!])$, $\Sigma' = \Sigma \cup \{\mathbf{sOld} \mapsto \mathbf{s}\}$, and $\Sigma'' = \Sigma \cup$

$\{\mathbf{sOld} \mapsto \mathbf{s}, \mathbf{sNew} \mapsto \mathbf{s}'\}$. According to the induction hypothesis it is equivalent to $(\forall\ i \in Ind : \Sigma \models \mathbf{s} \overset{\mathbf{op}(exp_1[i],\ldots,exp_n[i]) \blacktriangleleft \mathbf{Inst_{op}}[i]}{\rightarrow} \mathbf{s}') \wedge \varphi$. According to the operational semantic it is equivalent to $\Sigma \models s \overset{\overset{f_{\mathbf{Inst}}}{\underset{i \in Ind}{\parallel}} (\mathbf{op}(exp_1[i],\ldots,exp_n[i]))}{\rightarrow} s'$. It is equivalent to $\Sigma \models s \overset{\mathbf{t}}{\rightarrow} s'$.

□

Definition 56 (Equivalence relation on **SLS** terms)**.** *We say that two **SLS** terms $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$, which depend on the same free variables $v_1, \ldots, v_n$, are equivalent if and only if for each environment $\Sigma = \{v_1 \mapsto val_1, \ldots, v_n \mapsto val_n\}$ $\boldsymbol{t}_1 \overset{\Sigma}{=} \boldsymbol{t}_2$. We denote equivalence of terms $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$ as $\boldsymbol{t}_1 = \boldsymbol{t}_2$.*

Lemma 22. *"=" is an equivalence relation.*

Proof. The proof follows immediately from **Definition 56** and **Lemma 20**. □

Lemma 23. *For arbitrary sets A, B, A', and B' if*

$$A \ \sharp \ B \tag{6.1}$$
$$A \ \sharp \ B' \tag{6.2}$$
$$A \ \cup \ B = \mathbb{U} \tag{6.3}$$
$$A' \ \sharp \ B' \tag{6.4}$$
$$A' \ \sharp \ B \tag{6.5}$$
$$A' \ \cup \ B' = \mathbb{U} \tag{6.6}$$

then $A = A'$ and $B = B'$.

Proof. Let us first proof that $A = A'$.
$\forall o : o \in A \overset{(6.2)}{\Rightarrow} o \notin B' \overset{(6.4),(6.6)}{\Rightarrow} o \in A'$ which implies $A \subseteq A'$ (*).
$\forall o : o \in A' \overset{(6.5)}{\Rightarrow} o \notin B \overset{(6.1),(6.3)}{\Rightarrow} o \in A$ which implies $A' \subseteq A''$ (**).
$(*) \wedge (**) \Rightarrow A = A'$ (***).
Let us now proof that $B = B'$. $(6.3) \wedge (6.6) \Rightarrow A \cup B = A' \cup B' \overset{(***)}{\Rightarrow}$ $(A \cup B) \setminus A = (A' \cup B') \setminus A' \overset{(6.1),(6.4)}{\Rightarrow} B = B'$. □

Lemma 24. *If $\boldsymbol{SSOR}_1 \asymp \boldsymbol{SSOR}_2$ then $\boldsymbol{SSOR}_1 = \boldsymbol{SSOR}_2$.*

Proof. We have to prove that for any stereotype slices $\mathbf{s}$ and $\mathbf{s}'$ and environment Σ, $\Sigma' \models \mathbf{SysInv}[\mathbf{sOld}]$ implies $\Sigma' \models \mathbf{pre}_1 \wedge \mathbf{post}_1$ holds if and only if $\Sigma' \models \mathbf{pre}_2 \wedge \mathbf{post}_2$ holds, where $\Sigma' = \Sigma \cup \{\mathbf{sOld} \mapsto \mathbf{s}, \mathbf{sOld} \mapsto \mathbf{s}'\}$.

According to **Definition 48**, $\mathbf{SSOR}_1 \asymp \mathbf{SSOR}_2$ and $\Sigma' \models \mathbf{SysInv}[\mathbf{sOld}]$ implies $\Sigma' \models \mathbf{pre}_1$ holds if and only if $\Sigma' \models \mathbf{pre}_2$ holds. Therefore we can reduce the proof to the flowing statement $\Sigma' \models$

$\mathbf{SysInv}[\mathbf{sOld}] \wedge \mathbf{pre}_1$ implies $\Sigma' \models \mathbf{post}_1$ holds if and only if $\Sigma' \models \mathbf{post}_2$ holds.

We check the equivalence of the post conditions separately for each stereotype item "*it*". Let us denote a set of objects whose values of "*it*" are updated by $\mathbf{SSOR}_1$ as A, and a set of objects whose values are preserved by $\mathbf{SSOR}_1$ as B. We denote the corresponding sets for $\mathbf{SSOR}_2$ as A' and B'. From the definition of $\asymp$ we know that $\mathbf{SSOR}_1$ and $\mathbf{SSOR}_2$ are both consistent and deterministic which implies $A \sharp B$, $A \cup B = \mathbb{U}$, $A' \sharp B'$, $A' \cup B' = \mathbb{U}$. On the other hand in the definition of $\asymp$ we explicitly check that $A \sharp B'$ and $A' \sharp B$. By putting it all together and applying **Lemma 23** we infer that $A = A'$ and $B = B'$. By this we conclude that $\mathbf{SSOR}_1$ and $\mathbf{SSOR}_2$ update and preserve exactly the same sets of objects.

Nevertheless it is possible that updated objects get different values according to $\mathbf{SSOR}_1$ and $\mathbf{SSOR}_2$. The corresponding part of $\asymp$ guarantees that this does not happen. It checks that if a stereotype item of an object is affected by both a transformation rule of $\mathbf{SSOR}_1$ and a transformation rule of $\mathbf{SSOR}_2$ then the new values of the stereotype item are equal. By this we guarantee consistency of transformation rules and conclude the proof. □

Lemma 25. *For each environment Σ if $\boldsymbol{SSOR}_1 \overset{\Sigma}{=} \boldsymbol{SSOR}_2$ (1) and $\boldsymbol{SSOR}_1$ preserves the system invariant for Σ (2) then $\boldsymbol{SSOR}_2$ also preserves the system invariant for Σ.*

Proof. For each $\mathbf{s}$, $\mathbf{s}'$, Σ:

$$\begin{aligned}
&\Sigma' \models \mathbf{SysInv}[\mathbf{sOld}] \wedge \mathbf{pre}_{\mathbf{SSOR}_2} \wedge \mathbf{post}_{\mathbf{SSOR}_2} \overset{(1)}{\Rightarrow}\\
&\Sigma' \models \mathbf{SysInv}[\mathbf{sOld}] \wedge \mathbf{pre}_{\mathbf{SSOR}_1} \wedge \mathbf{post}_{\mathbf{SSOR}_1} \overset{(2)}{\Rightarrow}\\
&\Sigma' \models \mathbf{SysInv}[\mathbf{sNew}]
\end{aligned}$$

where $\Sigma' = \{\mathbf{sOld} \mapsto \mathbf{s}, \mathbf{sNew} \mapsto \mathbf{s}'\}$. □

Lemma 26. *If $\boldsymbol{SSOR}_1 = \boldsymbol{SSOR}_2$ and $\boldsymbol{SSOR}_1$ preserves the system invariant then $\boldsymbol{SSOR}_2$ preserves the system invariant.*

Proof. The desired property immediately follows from **Lemma 25**. □

6.4 SLS-to-SSOR translation

The operational semantics gives a good intuition regarding the behavior of an **SLS** term. However, due to its operational nature, we can't directly use it as a specification. We have to translate an **SLS** term into a pair of pre and post conditions in first order logic. As we mentioned before we achieve this goal by translating an **SLS** term into a corresponding **SSOR**. As soon as we have an **SSOR** we can construct its logical characterization as described in **Definition 40**.

Since we would like the resulting logical characterization to be consistent and preserve the system invariant, we construct a translation procedure in such a way that it always produces a proper **SSOR**.

We define the translation of **SLS** terms into **SSOR** in an inductive way. First we define how to construct a corresponding **SSOR** for a basic stereotype operation call. This is the base of the induction. Then we show how to construct a corresponding **SSOR** for each operation composer under an assumption that we know how to do it for subterms. We conclude our translation description by considering recursive operations. Here we use a user-provided inductive hypothesis to generate a corresponding **SSOR**. We use one more nested induction to prove correspondence between the constructed **SSOR** and the recursive operation. By this we conclude the inductive step of our translation.

The formal definition of a correspondence between an **SLS** term and an **SSOR** is the following:

Definition 57 (Correspondence between an **SLS** term and an **SSOR**)**.** *We say that an **SSOR** corresponds to an **SLS** term **t** if and only if they are equal as **SLS** terms:* $\boldsymbol{SSOR} = \boldsymbol{t}$.

Definition 58 (**SLS**-to-**SSOR** translation)**.** *We denote the result of the **SLS**-to-**SSOR** translation of an **SLS** term **t** as* $\boldsymbol{SSOR_t}$. *We define it by structural induction over **SLS** terms in the following way:*

- *the induction base cases are:*
  - *if an **SLS** term is equal to an **SSOR** then we do not need to translate it.*
  - *the translation of* `skip` *is denoted by* $\boldsymbol{SSOR_{skip}}$. *It is defined in **Definition 72**.*
  - *the translation of a basic operation call* $\boldsymbol{op}(exp_1, \ldots, \mathrm{exp}_n)$ *is denoted by* $\boldsymbol{SSOR_{op(\mathit{exp}_1,\ldots,\mathrm{exp}_n)}}$. *It is defined in **Definition 63**.*
  - *the translation of a composite recursive operation call* $\boldsymbol{op}(exp_1, \ldots, \mathrm{exp}_n)$ *is denoted by* $\boldsymbol{SSOR_{op(\mathit{exp}_1,\ldots,\mathrm{exp}_n)}}$. *It is defined in **Definition 71**. We treat a composite recursive operation call as a base case rather then an inductive case for the following reason. To define the translation we use only the transformation and frame rules of the operation but not the body. The body is used only to verify that the call has desired properties which we consider below. Therefore we do not need the translation of the operation body to translate the operation call.*
- *the induction step cases are:*

- *the translation of a composite nonrecursive operation call* $\boldsymbol{op}(exp_1, \ldots, \mathrm{exp}_n)$ *is denoted by* $\boldsymbol{SSOR}_{\boldsymbol{op}(exp_1,\ldots,\mathrm{exp}_n)}$*. It is defined in **Definition 64**. The translation is based on the translation of the operation body* $\boldsymbol{t_{op}}$*.*
- *the translation of an if statement* $\boldsymbol{if}\ (\varphi)\ \boldsymbol{t}_1\ \mathtt{else}\ \boldsymbol{t}_2$ *is denoted by* $\boldsymbol{SSOR}_{\boldsymbol{if}\ (\varphi)\ \boldsymbol{t}_1\ \mathtt{else}\ \boldsymbol{t}_2}$*. It is defined in **Definition 73**. The translation is based on the translation of the **SLS** terms* $\boldsymbol{t}_1$ *and* $\boldsymbol{t}_2$*.*
- *the translation of a sequential composition* $\boldsymbol{t}_1; \boldsymbol{t}_2$ *is denoted by* $\boldsymbol{SSOR}_{\boldsymbol{t}_1;\boldsymbol{t}_2}$*. It is defined in **Definition 81**. The translation is based on the translation of the **SLS** terms* $\boldsymbol{t}_1$ *and* $\boldsymbol{t}_2$*.*
- *the translation of a parallel composition* $\overset{f_{\boldsymbol{Inst}}}{\underset{i \in Ind}{\|}} \boldsymbol{op}(exp_1[i], \ldots, \mathrm{exp}_n[i])$ *is denoted by* $\boldsymbol{SSOR}_{\overset{f_{\boldsymbol{Inst}}}{\underset{i \in Ind}{\|}} \boldsymbol{op}(exp_1[i],\ldots,\mathrm{exp}_n[i])}$*. It is defined in **Definition 82**. The translation is based on the translation of the calls* $\boldsymbol{op}(exp_1[i], \ldots, exp_n[i])$*.*

The translation described in **Definition 58** does not terminate if there is an undesired cyclic dependency between the operations. Therefore below we introduce syntactical constraints which guarantee termination of the **SLS**-to-**SSOR** translation. To define this syntactical constraint we need the following auxiliary definitions.

Definition 59 (Order on stereotype operations)**.** *We define an order* $>$ *on stereotype operations in the following way. For stereotype operations* f *and* g*,* $f > g$ *if and only if:*

- f *is a composite operation,*
- *the body of* f *contains a call of* g*, and*
- *if* f *is a composite recursive operation then* $f \neq g$*.*

Definition 60. *We denote the set of all **SLS** terms which depend on stereotype operations from the set* F *and do not contain **SSOR** sub-terms as* $\boldsymbol{Trm}(F)$*.*

The syntactical constraint which guarantees termination of the **SLS**-to-**SSOR** translation is acyclicity of the order $>$ on stereotype operations. The relation $>$ can be constructed by visiting the bodies of all stereotype operations. And therefore its acyclicity can also be checked. Let us prove that the constraint is sufficient to guarantee termination of the **SLS**-to-**SSOR** translation.

Lemma 27. *If for any operation call of any operation which belongs to a set of stereotype operations F the translation of the operation call terminates then the translation of any* ***SLS*** *term from* ***Trm***(F) *also terminates.*

Proof. We prove the desired property by structural induction over **SLS** terms in the following way:

- the induction base cases are:
  - the translation of `skip`, basic operation calls, and recursive operation calls are defined in **Definition 72**, **Definition 63**, and **Definition 71**, respectively. The translation of each of them does not depend on other terms and therefore terminates.
- the induction step cases are:
  - the termination of the translation of a composite nonrecursive operation call follows from the lemma assumption.
  - the translation of the if statement, the sequential composition, and the parallel composition are defined in **Definition 73**, **Definition 81**, and **Definition 82**. The translation of each of them terminates if the translation of its sub-terms terminates. According to the inductive hypothesis the translation of the sub-terms terminates.

$\square$

Theorem 28 (**SLS**-to-**SSOR** translation terminates)**.** *For any set of stereotype operations F if the order $>$ on stereotype operations is acyclic on F then the for each* ***SLS*** *term from* ***Trm***(F)*, the* ***SLS****-to-****SSOR*** *translation of the term terminates.*

Proof. We prove the desired property by induction over the order $>$ on stereotype operations.

In the induction base we have to prove that the **SLS**-to-**SSOR** translation terminates on $\mathbf{Trm}(\varnothing)$. There are no stereotype operations which belong to $\varnothing$. Therefore according to **Lemma 27** the translation terminates on $\mathbf{Trm}(\varnothing)$.

In the induction step we know that the translation terminates for $\mathbf{Trm}(F')$, where $F' \subset F$. We have to prove that the translation terminates for $\mathbf{Trm}(F' \cup \{f\})$, where $f \in F \setminus F'$ and f's definition depends only on operations from F'.

If f is a basic or a composite recursive operation then according to **Definition 63** and **Definition 71** the translation of any call of the operation f terminates (1). Let us consider the case when f is a composite nonrecursive operation. Since the definition f depends only on operation calls of operations from F' the body of f belongs to $\mathbf{Trm}(F')$. Therefore according to the induction hypothesis the translation of the body of f terminates. Therefore according to **Definition 71** the translation of any call of the operation f terminates (2). From (1) and (2) we conclude that any call of the operation f terminates. From the induction hypothesis we know that the translation of any call of any operation from F' terminates. Therefore we can conclude that the translation of any call of any operation from $F' \cup \{f\}$ terminates. Therefore according to **Definition 27** the translation of any **SLS** term from $\mathbf{Trm}(F' \cup \{f\})$ terminates. □

Now we can formulate the main property of the **SLS**-to-**SSOR** translation. The property which we want to prove is the following one. For each $\mathbf{t}$, $\mathbf{SSOR_t}$ corresponds to $\mathbf{t}$ (see **Definition 57**) and is proper (see **Definition 45**). This property holds only if for each stereotype operation certain proof obligations hold. We will define the proof obligations in the following definitions:

- for basic operations in **Definition 61**.
- for stereotype constructors in **Definition 62**.
- for composite nonrecursive operations there are no proof obligations.
- for composite recursive operations **Definition 70**.

Lemma 29. *If for any operation call of any operation which belongs to a set of stereotype operations F the translation of the call is proper and corresponds to the operation call then the translation of any **SLS** term from **Trm**(F) is proper and corresponds to the term.*

Proof. We prove the desired property by structural induction over **SLS** terms in the following way:

- the induction base cases are:
  - according to **Lemma 44** $\mathbf{SSOR}_{\texttt{skip}}$ is proper. According to **Lemma 45** $\mathbf{SSOR}_{\texttt{skip}}$ corresponds to `skip`.
  - according to the lemma's assumption the translation of a basic operation call or a recursive operation call is proper and corresponds to the basic operation call and recursive operation call, respectively.
- the induction step cases are:
  - according to the lemma's assumption the translation of a nonrecursive operation call is proper and corresponds to the operation call.

– According to **Lemma 46** the translation of an if statement is proper and corresponds to the if statement if the translation of the sub-terms are proper and correspond to the sub-terms. The validity of the last statement follows from the induction hypothesis.

– According to **Lemma 56** and **Lemma 57** the translation of a sequential composition is proper and corresponds to the sequential composition if the translation of the sub-terms are proper and correspond to the sub-terms. The validity of the last statement follows from the induction hypothesis.

– According to **Lemma 58** and **Lemma 59** the translation of a parallel composition is proper and corresponds to the parallel composition if the translation of the sub-terms are proper and correspond to the sub-terms. The validity of the last statement follows from the induction hypothesis.

□

Theorem 30 (Properties of **SLS**-to-**SSOR** translation)**.** *For any set of stereotype operations F if the order $>$ on stereotype operations is acyclic on F and the proof obligations hold for each operation from F then for each **SLS** term from **Trm**(F) the **SLS**-to-**SSOR** translation of the term is proper and corresponds to the term.*

Proof. We prove the desired property by induction over the order $>$ on stereotype operations.

In the induction base we have to prove that the **SLS**-to-**SSOR** translation of any term from **Trm**$(\varnothing)$ is proper and corresponds to the term. There are no stereotype operations which belong to $\varnothing$. Therefore according to **Lemma 29** the translation of any term from **Trm**$(\varnothing)$ is proper and corresponds to the term.

In the induction step we know that the translation of any term from **Trm**(F') is proper and correspond to the term, where $F' \subset F$. We have to prove that the translation of any term from **Trm**$(F' \cup \{f\})$ is proper and corresponds to the term, where $f \in F \setminus F'$ and f's definition depends only on operations from F'.

If f is a basic operation then according to **Lemma 35** and **Lemma 36** since the operation proof obligations hold the translation of any call of the operation f is proper and corresponds to the operation call (1).

Let us consider the case when f is a recursive composite operation. The definition f depends only on operation calls of operations from F' and on f. Therefore the body of f where all recursive calls are replaced by **SSOR** belongs to **Trm**(F'). Therefore according to the induction hypothesis the translation of the modified body of f is proper and corresponds to the body.

Therefore according to **Lemma 43** since the operation proof obligations holds the translation of any call of the operation f is proper and corresponds to the operation call (2).

Let us consider the case when f is a nonrecursive composite operation. Since the definition f depends only on operation calls of operations from F' the body of f belongs to $\mathbf{Trm}(F')$. Therefore according to the induction hypothesis the translation of the body of f is proper and corresponds to the body. Therefore according to **Lemma 37** and **Lemma 38** since the operation proof obligations hold the translation of any call of the operation f is proper and corresponds to the operation call (3).

From (1), (2), and (3) we conclude that the translation of any call of the operation f is proper and corresponds to the operation call. From the induction hypothesis we know that the translation of any call of any operation from F' is proper and corresponds to the operation call. Therefore we can conclude that the translation of any call of any operation from $F' \cup \{f\}$ is proper and corresponds to the operation call. Therefore according to **Lemma 29** the translation of any **SLS** term from $\mathbf{Trm}(F' \cup \{f\})$ is proper and corresponds to the term. □

The rest of the section is organized in the following way. In **Subsection 6.4.1** we define the translation and prove the properties of the basic operation calls. In **Subsection 6.4.2** we define the translation and prove the properties of both the recursive and the nonrecursive composite operation calls. In **Subsection 6.4.3** we define the translation and prove the properties of `skip`. In **Subsection 6.4.4** we define the translation and prove the properties of the conditional statement. In **Subsection 6.4.5** we define the translation and prove the properties of the sequential composition. In **Subsection 6.4.6** we define the translation and prove the properties of the parallel composition.

6.4.1 Basic operation call

Let us now consider the proof obligations which guarantee that a basic operation has the desired properties.

Definition 61 (Proof obligations for a basic operation)**.** *For a basic operation* $\boldsymbol{op}(v_1 : T_1, \ldots, v_n : T_n)$ *under assumption* $\boldsymbol{SysInv}[\boldsymbol{sOld}] \wedge \boldsymbol{pre_{op}}$ *we check the following proof obligations:*

1. *for each* i *in* $[1..n^{in}]$ $exp_i^{in} \neq \boldsymbol{null}$ *and* $exp_i^{in}.\mathtt{instID} = exp_i^{in}$
2. *for each disjoint* i *and* j *in* $[1..n^{in}]$ $exp_i^{in} \neq exp_j^{in}$
3. *for each* i *in* $[1..n^{out}]$ $exp_i^{out} \neq \varnothing$
4. *for each disjoint* i *and* j *in* $[1..n^{out}]$ $exp_i^{out} \sharp exp_j^{out}$

5. $exp_1^{in}.\mathrm{Elements} \cup \ldots \cup exp_{n_{in}}^{in}.\mathrm{Elements} = exp_1^{out} \cup \ldots \cup exp_{n_{in}}^{out}$

6. *for each disjoint* i *and* j *in* $[1..n_t]$ $S_i^t \sharp S_j^t$

7. *for each* i *in* $[1..n_t]$ $S_i^t \subset exp_1^{in}.\mathrm{Elements} \cup \ldots \cup exp_{n_{in}}^{in}.\mathrm{Elements}$

8. *For each transformation rule* $\langle \psi_{S^t}^-, it^t, val \rangle$ *where* it^t *is a stereotype item of type* Reg *we check* $\forall \boldsymbol{v} : \psi_{S^t}^- \Rightarrow val \subseteq exp_1^{in}.\mathrm{Elements} \cup \ldots \cup exp_{n_{in}}^{in}.\mathrm{Elements}$.

9. *For each transformation rule* $\langle \psi_{S^t}^-, it^t, val \rangle$ *where* it^t *is a stereotype item of type* ref *we check* $\forall \boldsymbol{v} : \psi_{S^t}^- \Rightarrow val \in exp_1^{in}.\mathrm{Elements} \cup \ldots \cup exp_{n_{in}}^{in}.\mathrm{Elements}$.

10. *under the additional assumption* $\boldsymbol{post_{op}} \wedge \boldsymbol{SysInvEl}[\boldsymbol{sNew}]$ *for each* i *in* $[1..n^{out}]$ *we check* $\boldsymbol{Inv}_{\mathrm{St}}[\boldsymbol{sNew}, exp_i^{out}]$

where:

- exp_i^{in} *for* i *from* $[1..n_{in}]$ *are expressions of type* ref *which contain identifiers of affected instances.*
- exp_i^{out} *for* i *from* $[1..n_{in}]$ *are expressions of type* Reg *which contain the elements of the output instances.*
- $\langle \psi_{S_1^t}^-, it_1^t, val_1 \rangle, \ldots, \langle \psi_{S_{n_t}^t}^-, it_{n_t}, val_{n_t} \rangle$ *are the transformation rules of the operation.*

You can see that we completely ignore **SysInvID**[**sNew**] in our proof obligations. It is because we check that it is implied by a stereotype invariant and **SysInvEl**. Under this assumption it is enough to prove that the stereotype invariant and **SysInvEl** are preserved by a stereotype operation to prove the preservation of the system invariant.

Lemma 31 (Properties of an $\mathbf{SSOR_{op}}$ of a basic operation)**.** *If the proof obligations from **Definition 61** hold then* $\boldsymbol{SSOR_{op}}$ *of a basic stereotype operation* $\boldsymbol{op}(v_1 : T_1, \ldots, v_n : T_n)$ *is proper (consistent, deterministic, and preserves the system invariant) and affects only stereotype instances from* $\boldsymbol{Inst}_{op}$.

Proof. Let us prove the desired properties one by one:

- **Consistency:** Let us first prove consistency between transformation rules. There are two kinds of transformation rules: user provided and generated. Since the generated transformation rule updates the *Elements* stereotype item and the user provided transformation rule doesn't there is no inconsistency between them. The proof obligation (4) guarantees consistency of the user provided transformation rules. The proof obligation (6) guarantees consistency of the generated transformation rules. Since frame rules are constructed as negations of transformation rules there is no inconsistency between them either.

- **Determinism:** Outside of affected instances the operation preserves all values in a deterministic way. Inside of the affected instances we have the same two kinds of updates: user provided and generated. The user provided specifications are deterministic because of the way in which we constructed the frame rules. Each object belongs to a set or to a set complement. The generated transformation rules are deterministic because of the proof obligations (5) and (4). Each object of the participating instances is affected by exactly one generated transformation rule.

- **Affected instances:** The proof obligation (7) guarantees that user provided transformation rules affect only stereotype instances from $\mathbf{Inst_{op}}$. The proof obligation (5) does the same for the generated transformation rules. The proof obligations (1) and (2) guarantee that $\mathbf{Inst_{op}}$ contains only instance IDs and doesn't contain **null**.

- $\mathbf{SysInvEl}[\mathbf{sNew}]$**:** Since we know that the operation affects only instances form $\mathbf{Inst_{op}}$ it is enough to check invariant preservation only for these instances. The proof obligations (3) to (5) and the way how we generate transformation rules guarantee preservation of the first two properties of **SysInvEl**. The proof obligations (8) and (9) establish the third and forth property of **SysInvEl**, respectively.

- $\mathbf{Inv}_{St}[\mathbf{sNew}]$**:** Similarly to the previous paragraph it is enough to check that the stereotype invariant is preserved by instances from $\mathbf{Inst_{op}}$. The proof obligation (10) checks exactly this property.

- $\mathbf{SysInv}[\mathbf{sNew}]$**:** Since we already know that $\mathbf{SysInvEl}[\mathbf{sNew}]$ and $\mathbf{Inv}_{St}[\mathbf{sNew}]$ hold, we can infer from the stereotype proof obligation that $\mathbf{SysInvID}[\mathit{StSlice}]$ also holds, which implies that $\mathbf{SysInv}[\mathbf{sNew}]$ holds.

□

Definition 62 (Proof obligations for a constructor)**.** *For a stereotype constructor* `constructor` $\boldsymbol{op}(obj : \mathrm{ref!}, v_1 : T_1, \dots, v_n : T_n)$ *we check the following proof obligation:*
$\boldsymbol{SysInv}[\boldsymbol{sOld}] \wedge \boldsymbol{post_{op}} \wedge \boldsymbol{SysInvEl}[\boldsymbol{sNew}] \Rightarrow \boldsymbol{Inv}_{\mathrm{St}}[\boldsymbol{sNew}, \{obj\}]$.

Lemma 32 (Properties of an $\mathbf{SSOR_{op}}$ of a constructor)**.** *If the proof obligation from* ***Definition 62*** *holds then the* $\boldsymbol{SSOR_{op}}$ *of a stereotype constructor* ***`constructor`*** ***op***$(obj : \text{ref!}, v_1 : T_1, \ldots, v_n : T_n)$ *is proper and effects only stereotype instance* $\{obj\}$*.*

Proof. Let us prove the desired properties one by one:

- **Consistency:** There are two kinds of transformation rules: user provided and generated. Since the generated transformation rule updates the *Elements* stereotype item and the user provided transformation rule doesn't there is no inconsistency between them. The syntactical constraint (1) from **Definition 51** guarantees consistency of the user provided transformation rules. Since there is only one generated rule it is consistent. Since the only frame rule preserves only values of objects which are not equal to *obj* it is consistent with the transformation rule.

- **Determinism:** Outside $\{obj\}$ the constructor in a deterministic way preserves all values. According to the syntactical constraint (1) from **Definition 51** there is exactly one transformation rule for each stereotype item. This transformation rule defines an update of stereotype instance $\{obj\}$ in a deterministic way.

- **Affected instances:** The shape of transformation rules guarantees that they affect only obj.

- **SysInvEl**[**sNew**]**:** Since we know that the constructor affects only *obj* it is enough to check invariant preservation only for the instance $\{obj\}$. The way how we generate transformation rules guarantees preservation of the first two properties of **SysInvEl**. The syntactical constraints (2), (3), and (4) from **Definition 51** establish the third and forth property of **SysInvEl**, respectively.

- $\mathbf{Inv}_{St}$[**sNew**]**:** Similarly to the previous paragraph it is enough to check that the stereotype instance $\{obj\}$ satisfies the stereotype invariant. The corresponding proof obligation checks exactly this property.

- **SysInv**[**sNew**]**:** Since we already know that **SysInvEl**[**sNew**] and $\mathbf{Inv}_{St}$[**sNew**] hold we can infer from the stereotype proof obligation that **SysInvID**[*StSlice*] also holds, which implies that **SysInv**[**sNew**] holds.

□

Here we assume that allocation of a new object *obj* happens at the beginning of the constructor execution. Or, in other words, a stereotype slice **sNew** contains object *obj* and a stereotype slice **sOld** doesn't. Therefore when we state that **SysInv**[**sOld**] holds we say nothing about the $\{obj\}$ stereotype instance. To prevent usage of values of stereotype items of the object *obj* in a stereotype slice **sOld**, which are undefined, we introduced the syntactical constraint (5) from **Definition 51**.

Before we define an **SSOR** for a basic operation call we have to prove the following auxiliary lemmas.

Lemma 33. *For each expression φ which depends only on variables $v_1, \ldots, v_n$, expressions $exp_1, \ldots, exp_n$ which depend only on free variables $v'_1, \ldots, v'_{n'}$, and an environment $\Sigma = \{v'_1 \mapsto val_1, \ldots, v'_{n'} \mapsto val_{n'}\}$ the following holds: $[\![exp]\!]_{\Sigma'} = [\![exp']\!]_{\Sigma}$, where $\Sigma' = \{v_1 \mapsto [\![exp_1]\!]_{\Sigma}, \ldots, v_n \mapsto [\![exp_n]\!]_{\Sigma}\}$ and $exp' = exp[v_1 := exp_1, \ldots, v_n := exp_n]$.*

Proof. We prove the lemma by structural induction over expressions. The induction base consists of the following two cases:

- IB$_1$: *exp* does not depend on variables.
- IB$_2$ *exp* is equal to v_{i_0} where $i_0 \in [1..n]$.

Let us prove IB$_1$. In this case *exp* does not depend on variables (1). (1) implies $exp = exp'$ (2). $[\![exp]\!]_{\Sigma'} \overset{(1)}{=} [\![exp]\!]_{\Sigma} \overset{(2)}{=} [\![exp']\!]_{\Sigma}$.

Let us prove IB$_2$. In this case $exp = v_{i_0}$ (3) where $i_0 \in [1..n]$. $exp' = exp[v_1 := exp_1, \ldots, v_n := exp_n] \overset{(3)}{=} v_{i_0}[v_1 := exp_1, \ldots, v_n := exp_n] = exp_{i_0}$ (4). $[\![exp]\!]_{\Sigma'} \overset{(3)}{=} [\![v_{i_0}]\!]_{\Sigma'} = [\![exp_{i_0}]\!]_{\Sigma} \overset{(4)}{=} [\![exp']\!]_{\Sigma}$.

Let us now prove the induction step. In this case $exp = f(exp_1^{sub}, \ldots, exp_m^{sub})$ (5), where f is a function of the arity m and $exp_1^{sub}, \ldots, exp_m^{sub}$ are expressions. We denote $exp_j^{sub}[v_1 := exp_1, \ldots, v_n := exp_n]$ as ${exp'}_j^{sub}$ where $j \in [1..n]$. According to the induction hypothesis the following holds. For each $j \in [1..m]$ $[\![exp_j^{sub}]\!]_{\Sigma'} = [\![{exp'}_j^{sub}]\!]_{\Sigma}$ (6). We denote interpretation of the function f as $[\![f]\!]$. $exp' = exp[v_1 := exp_1, \ldots, v_n := exp_n] \overset{(5)}{=} f(exp_1^{sub}, \ldots, exp_m^{sub})[v_1 := exp_1, \ldots, v_n := exp_n] = f({exp'}_1^{sub}, \ldots, {exp'}_m^{sub})$ (7).

$$[\![exp]\!]_{\Sigma'} \overset{(5)}{=} [\![f(exp_1^{sub}, \ldots, exp_m^{sub})]\!]_{\Sigma'} = [\![f]\!]([\![exp_1^{sub}]\!]_{\Sigma'}, \ldots, [\![exp_m^{sub}]\!]_{\Sigma'}) \overset{(6)}{=}$$
$$[\![f]\!]([\![{exp'}_1^{sub}]\!]_{\Sigma}, \ldots, [\![{exp'}_m^{sub}]\!]_{\Sigma}) = [\![f({exp'}_1^{sub}, \ldots, {exp'}_m^{sub})]\!]_{\Sigma} \overset{(7)}{=} [\![exp']\!]_{\Sigma}$$

□

Lemma 34. *For each formula φ, expressions $exp_1, \ldots, exp_n$, and an environment $\Sigma = \{v'_1 \mapsto val_1, \ldots, v'_{n'} \mapsto val_{n'}\}$ if:*

- *φ depends only on free variables $v_1, \ldots, v_n$*

- *φ does not contain quantifiers which quantify over v_i where $i \in [1..n]$*
- *expressions $exp_1, \ldots, exp_n$ depends only on free variables $v'_1, \ldots, v'_{n'}$*

then the following holds:
$\Sigma' \models \varphi \Leftrightarrow \Sigma \models \varphi'$, where $\Sigma' = \{v_1 \mapsto [\![exp_1]\!]_\Sigma, \ldots, v_n \mapsto [\![exp_n]\!]_\Sigma\}$ and $exp' = \varphi[v_1 := exp_1, \ldots, v_n := exp_n]$.

Proof. We prove the lemma by structural induction over formulas.

Let us prove the induction base. In this case $\varphi = P(exp_1^{sub}, \ldots, exp_m^{sub})$. The proof is identical to the proof of the induction step of **Definition 33**. Therefore we omit the proof.

Let us prove the induction step. In this case $\varphi = \rho(\varphi_1, \ldots, \varphi_m)$ (1) where ρ is a logical binder or a quantifier which depends on m sub-formulas. We denote $\varphi_j[v_1 := exp_1, \ldots, v_n := exp_n]$ as ${\varphi'}_j$ where $j \in [1..n]$. According to the induction hypothesis the following holds. For each $j \in [1..m]$ $\Sigma' \models \varphi_j \Leftrightarrow \Sigma \models {\varphi'}_j$ (2). We denote interpretation of ρ as $[\![\rho]\!]$. $\varphi' = \varphi[v_1 := exp_1, \ldots, v_n := exp_n] \overset{(1)}{=} \rho(\varphi_1, \ldots, \varphi_m)[v_1 := exp_1, \ldots, v_n := exp_n] = \rho({\varphi'}_1, \ldots, {\varphi'}_m)$ (3).

$$[\![\varphi]\!]_{\Sigma'} \overset{(1)}{\Leftrightarrow} [\![\rho(\varphi_1, \ldots, \varphi_m)]\!]_{\Sigma'} \Leftrightarrow [\![\rho]\!](\Sigma' \models \varphi_1, \ldots, \Sigma' \models \varphi_m) \overset{(2)}{\Leftrightarrow}$$
$$[\![\rho]\!](\Sigma \models {\varphi'}_1, \ldots, \Sigma \models {\varphi'}_m) \Leftrightarrow [\![\rho({\varphi'}_1, \ldots, {\varphi'}_m)]\!]_\Sigma \overset{(3)}{\Leftrightarrow} \Sigma \models \varphi'$$ □

Definition 63 (**SSOR** for a call of a basic operation)**.** *We define an **SSOR** for a call **op**$(exp_1, \ldots, \mathrm{exp}_n)$ of a basic operation **op** as $\boldsymbol{SSOR_{op}}[v_1 := exp_1, \ldots, v_n := exp_n]$, where $\boldsymbol{SSOR_{op}}$ is an **SSOR** of a basic operation **op**. We denote it as $\boldsymbol{SSOR}_{\mathbf{op}(exp_1,\ldots,\mathrm{exp}_n)}$.*

Lemma 35. *$\boldsymbol{SSOR}_{\mathbf{op}(exp_1,\ldots,\mathrm{exp}_n)}$ corresponds to **op**$(exp_1, \ldots, \mathrm{exp}_n)$.*

Proof. For each Σ according to OS-Call-Basic $\Sigma \models \mathbf{s} \overset{\mathbf{op}(exp_1,\ldots,exp_n)}{\rightarrow} \mathbf{s'}$ holds if and only if $\Sigma'' \models \mathbf{SysInv}[\mathbf{sOld}] \wedge \mathbf{pre_{op}} \wedge \mathbf{post_{op}}$ holds, where $\Sigma'' = \{v_1 \mapsto [\![exp_1]\!]_{\Sigma'}, \ldots, v_n \mapsto [\![exp_n]\!]_{\Sigma'}, \mathbf{sOld} \mapsto \mathbf{s}, \mathbf{sNew} \mapsto \mathbf{s'}\}$ and $\Sigma' = \Sigma \cup \{\mathbf{sOld} \mapsto \mathbf{s}\}$. According to **Lemma 34** it is equivalent to $\Sigma''' \models (\mathbf{SysInv}[\mathbf{sOld}] \wedge \mathbf{pre_{op}} \wedge \mathbf{post_{op}})[v_1 := exp_1, \ldots, v_n := exp_n]$, where $\Sigma''' = \Sigma \cup \{\mathbf{sOld} \mapsto \mathbf{s}, \mathbf{sNew} \mapsto \mathbf{s'}\}$. Since the system invariant doesn't depend on $v_1, \ldots, v_n$ the last statement is equivalent to $\Sigma''' \models \mathbf{SysInv}[\mathbf{sOld}] \wedge (\mathbf{pre_{op}}[v_1 := exp_1, \ldots, v_n := exp_n]) \wedge (\mathbf{post_{op}}[v_1 := exp_1, \ldots, v_n := exp_n])$. According to the **Lemma 17** we get $\Sigma''' \models \mathbf{SysInv}[\mathbf{sOld}] \wedge \mathbf{pre}_{\mathbf{op}(exp_1,\ldots,\mathrm{exp}_n)} \wedge \mathbf{post}_{\mathbf{op}(exp_1,\ldots,\mathrm{exp}_n)}$ which concludes the proof of the desired property. □

Lemma 36. *If **op** is a basic operation and the proof obligations from **Definition 61** hold or if **op** is a constructor and the proof obligations from **Definition 62** hold then $\boldsymbol{SSOR}_{\mathbf{op}(exp_1,\ldots,\mathrm{exp}_n)}$ is proper.*

Proof. **Lemma 31** and **Lemma 32** imply that $\mathbf{SSOR_{op}}$ is proper. By applying **Lemma 19** we can infer that $\mathbf{SSOR}_{\mathbf{op}(exp_1,\ldots,\mathrm{exp}_n)}$ is also proper. □

6.4.2 Composite operation call

As we mentioned above there are two kinds of composite operations: recursive and nonrecursive. Let us first consider calls of composite nonrecursive operations.

Definition 64 (**SSOR** for a call of a composite nonrecursive operation)**.** *We define an **SSOR** for a call **op**($exp_1, \ldots, \mathrm{exp}_n$) of a composite nonrecursive operation **op** as $\boldsymbol{SSOR}_{\boldsymbol{t_{op}}}[v_1 := exp_1, \ldots, v_n := exp_n]$, where $\boldsymbol{t_{op}}$ is the body of the composite operation **op**. We denote it as $\boldsymbol{SSOR}_{\boldsymbol{op}(exp_1,\ldots,\mathrm{exp}_n)}$.*

Lemma 37. *$\boldsymbol{SSOR}_{\boldsymbol{op}(exp_1,\ldots,\mathrm{exp}_n)}$ corresponds to $\boldsymbol{op}(exp_1, \ldots, \mathrm{exp}_n)$.*

Proof. For each Σ according to OS-Call-Comp $\Sigma \models \mathbf{s} \overset{\mathbf{op}(exp_1,\ldots,exp_n)}{\to} \mathbf{s'}$ holds if and only if $\Sigma'' \models \mathbf{s} \overset{\mathbf{t_{op}}}{\to} \mathbf{s'}$ holds, where $\Sigma'' = \{v_1 \mapsto [\![exp_1]\!]_{\Sigma'}, \ldots, v_n \mapsto [\![exp_n]\!]_{\Sigma'}\}$, $\Sigma' = \Sigma \cup \{\mathbf{sOld} \mapsto \mathbf{s}\}$, and $\mathbf{t_{op}}$ is the body of **op**. According to the inductive hypothesis it is equivalent to $\Sigma''' \models \mathbf{SysInv}[\mathbf{sOld}] \wedge \mathbf{pre}_{\mathbf{t_{op}}} \wedge \mathbf{post}_{\mathbf{t_{op}}}$, where $\Sigma''' = \Sigma'' \cup \{\mathbf{sOld} \mapsto \mathbf{s}, \mathbf{sNew} \mapsto \mathbf{s'}\}$. According to **Lemma 34** it is equivalent to $\Sigma'''' \models (\mathbf{SysInv}[\mathbf{sOld}] \wedge \mathbf{pre}_{\mathbf{t_{op}}} \wedge \mathbf{post}_{\mathbf{t_{op}}})[v_1 := exp_1, \ldots, v_n := exp_n]$, where $\Sigma'''' = \Sigma \cup \{\mathbf{sOld} \mapsto \mathbf{s}, \mathbf{sNew} \mapsto \mathbf{s'}\}$. Since the system invariant doesn't depend on $v_1, \ldots, v_n$ the last statement is equivalent to $\Sigma'''' \models \mathbf{SysInv}[\mathbf{sOld}] \wedge (\mathbf{pre}_{\mathbf{t_{op}}}[v_1 := exp_1, \ldots, v_n := exp_n]) \wedge (\mathbf{post}_{\mathbf{t_{op}}}[v_1 := exp_1, \ldots, v_n := exp_n])$. According to the **Lemma 17** we get $\Sigma'''' \models \mathbf{SysInv}[\mathbf{sOld}] \wedge \mathbf{pre}_{\mathbf{op}(exp_1,\ldots,\mathrm{exp}_n)} \wedge \mathbf{post}_{\mathbf{op}(exp_1,\ldots,\mathrm{exp}_n)}$ which concludes the proof of the desired property. ☐

Lemma 38. *If the translation of the composite nonrecursive operation body $\boldsymbol{SSOR}_{\boldsymbol{t_{op}}}$ is proper and corresponds to $\boldsymbol{t_{op}}$ then $\boldsymbol{SSOR}_{\boldsymbol{op}(exp_1,\ldots,\mathrm{exp}_n)}$ is proper.*

Proof. According to **Definition 64** $\mathbf{SSOR}_{\mathbf{op}(exp_1,\ldots,\mathrm{exp}_n)} = \mathbf{SSOR}_{\mathbf{t_{op}}}[v_1 := exp_1, \ldots, v_n := exp_n]$. According to **Lemma 19** if $\mathbf{SSOR}_{\mathbf{t_{op}}}$ is proper then $\mathbf{SSOR}_{\mathbf{t_{op}}}[v_1 := exp_1, \ldots, v_n := exp_n]$. Therefore we conclude that $\mathbf{SSOR}_{\mathbf{op}(exp_1,\ldots,\mathrm{exp}_n)}$ is also proper. ☐

Let us now consider calls of composite recursive operations.

Definition 65 (**SSOR** for a composite recursive operation)**.** *We define an $\boldsymbol{SSOR}_{\boldsymbol{op}}$ for a composite recursive operation **op** in the following way:*

- *pre-condition $\boldsymbol{pre}_{\boldsymbol{op}}$.*
- *transformation rules are $\langle \psi^-_{S^t_1}, it^t_1, val_1 \rangle, \ldots, \langle \psi^-_{S^t_{n_t}}, it_{n_t}, val_{n_t} \rangle$.*
- *frame rules are $\langle \psi^-_{S^f_1}, it^f_1 \rangle, \ldots, \langle \psi^-_{S^f_{n_f}}, it_{n_f} \rangle$.*

To define the proof obligations for a composite recursive operation we need to introduce some auxiliary definitions and prove their properties.

Definition 66 (Properties of an **SSOR** on an environment Σ)**.** *We say that an* ***SSOR*** *is consistent, deterministic, preserves the system invariant, (or proper) on an environment* Σ *if and only if a corresponding property of the* ***SSOR*** *holds under an assumption* $v_1 = \Sigma[\![v_1]\!], \dots, v_n = \Sigma[\![v_n]\!]$*, where* $v_1, \dots, v_n$ *are variables on which the* ***SSOR*** *and the environment* Σ *depend.*

Definition 67 (Order over environments)**.** *We define an order over environments induced by an order over* $v_{\boldsymbol{mes}}$ *in the following way* $\Sigma < \Sigma' \Leftrightarrow \Sigma[\![v_{\boldsymbol{mes}}]\!] < \Sigma'[\![v_{\boldsymbol{mes}}]\!]$.

Definition 68 (**SSOR** for a recursive call)**.** *We define an* ***SSOR*** *for a recursive call* $\boldsymbol{op}(\mathrm{exp}_1, \dots, \mathrm{exp}_n)$ *of a composite recursive operation* ***op*** *as* $\boldsymbol{SSOR_{op}}[v_1 := \mathrm{exp}_1, \dots, v_n := \mathrm{exp}_n]$ *to whose pre-condition we add property* $\mathrm{exp}_{\boldsymbol{mes}} < v_{\boldsymbol{mes}}$*, where* $v_{\boldsymbol{mes}}$ *is the measure variable and* $\mathrm{exp}_{\boldsymbol{mes}}$ *is an expression which corresponds to the value of* $v_{\boldsymbol{mes}}$ *in the call. We denote it as* $\boldsymbol{SSOR}_{\boldsymbol{op}^{nr}(exp_1,\dots,\mathrm{exp}_n)}$.

Lemma 39. *For each* Σ *if* $\forall\ \Sigma' : \Sigma' < \Sigma \Rightarrow \boldsymbol{t_{op}} \overset{\Sigma'}{=} \boldsymbol{SSOR_{op}}$ *then* $\boldsymbol{op}(\mathrm{exp}_1, \dots, \mathrm{exp}_n) \overset{\Sigma}{=} \boldsymbol{SSOR}_{\boldsymbol{op}^{nr}(exp_1,\dots,\mathrm{exp}_n)}$.

Proof. For each $\mathbf{s}$, $\mathbf{s}'$, and Σ such that $[\![\mathrm{exp}_{\mathbf{mes}}]\!]_{\Sigma'} < \Sigma[\![v_{\mathbf{mes}}]\!]$, where $\Sigma' = \Sigma \cup \{\mathbf{sOld} \mapsto \mathbf{s}\}$, the proof is the same as for a nonrecursive case, see **Lemma 37**.

Let us consider the other case when $[\![\mathrm{exp}_{\mathbf{mes}}]\!]_{\Sigma'} < \Sigma[\![v_{\mathbf{mes}}]\!]$ does not hold. According to the operational semantic $\mathbf{op}(\mathrm{exp}_1, \dots, \mathrm{exp}_n)$ rejects any pair of states $\mathbf{s}$ and $\mathbf{s}'$ $(*)$. On the other hand the pre-condition of $\mathbf{SSOR}_{\mathbf{op}^{\mathrm{nr}}(exp_1,\dots,\mathrm{exp}_n)}$ does not hold for any state $\mathbf{s}$ $(**)$. $(*)$ and $(**)$ imply that $\mathbf{op}(\mathrm{exp}_1, \dots, \mathrm{exp}_n) \overset{\Sigma}{=} \mathbf{SSOR}_{\mathbf{op}^{\mathrm{nr}}(exp_1,\dots,\mathrm{exp}_n)}$. □

Lemma 40. *For each* Σ *if for each* Σ' *such that* $\Sigma' < \Sigma$*,* $\boldsymbol{SSOR_{op}}$ *is proper on* Σ' *then* $\boldsymbol{SSOR}_{\boldsymbol{op}^{nr}(exp_1,\dots,\mathrm{exp}_n)}$ *is proper on* Σ.

Proof. For each $\mathbf{s}$, $\mathbf{s}'$, and Σ such that $[\![\mathrm{exp}_{\mathbf{mes}}]\!]_{\Sigma'} < \Sigma[\![v_{\mathbf{mes}}]\!]$ $\mathbf{SSOR}_{\mathbf{op}^{\mathrm{nr}}(exp_1,\dots,\mathrm{exp}_n)}$ is proper on Σ if and only if $\mathbf{SSOR_{op}}$ is proper on Σ'', where $\Sigma' = \Sigma \cup \{\mathbf{sOld} \mapsto \mathbf{s}\}$ and $\Sigma'' = \{v_1 \mapsto [\![exp_1]\!]_{\Sigma'}, \dots, v_n \mapsto [\![exp_n]\!]_{\Sigma'}\}$. It is enough to prove the later property. Since $[\![\mathrm{exp}_{\mathbf{mes}}]\!]_{\Sigma'} < \Sigma[\![v_{\mathbf{mes}}]\!]$ implies $\Sigma'' < \Sigma$ we can apply the lemma pre-condition and get $\mathbf{SSOR_{op}}$ is proper on Σ''. In other case the pre-condition of the $\mathbf{SSOR}_{\mathbf{op}^{\mathrm{nr}}(exp_1,\dots,\mathrm{exp}_n)}$ doesn't hold on Σ which implies that $\mathbf{SSOR}_{\mathbf{op}^{\mathrm{nr}}(exp_1,\dots,\mathrm{exp}_n)}$ is proper on Σ. □

Definition 69 (Nonrecursive body of a recursive operation)**.** *We define a nonrecursive body for a composite recursive operation* ***op*** *by replacing in the body of the operation* $\boldsymbol{t_{op}}$ *all recursive calls* $\boldsymbol{op}(exp_1, \dots, \mathrm{exp}_n)$ *by* $\boldsymbol{SSOR}_{\boldsymbol{op}^{nr}(exp_1,\dots,\mathrm{exp}_n)}$*. We denote it as* $\boldsymbol{t^{nr}_{op}}$.

Lemma 41. *For each Σ if $\forall\ \Sigma' : \Sigma' < \Sigma \Rightarrow \boldsymbol{t_{op}} \overset{\Sigma'}{=} \boldsymbol{SSOR_{op}}$ and $\boldsymbol{SSOR_{op}} = \boldsymbol{SSOR_{t_{op}^{nr}}}$ then $\boldsymbol{t_{op}} \overset{\Sigma}{=} \boldsymbol{SSOR_{op}}$.*

Proof. **Lemma 21** and **Lemma 39** imply that by replacing a recursive call $\mathbf{op}(\exp_1, \dots, \exp_n)$ in $\mathbf{t_{op}}$ by $\mathbf{SSOR_{op^{nr}(\mathit{exp}_1,\dots,\exp_n)}}$ we get a Σ equivalent term. By applying this substitution for each recursive call we get $\mathbf{t_{op}} \overset{\Sigma}{=} \mathbf{t_{op}^{nr}}$. By applying the transitivity property of $\overset{\Sigma}{=}$ to the last observation and the equality $\mathbf{SSOR_{op}} = \mathbf{SSOR_{t_{op}^{nr}}}$ we get $\mathbf{t_{op}} \overset{\Sigma}{=} \mathbf{SSOR_{op}}$. □

Definition 70 (Proof obligations for a composite recursive operation)**.** *For a composite recursive operation $\boldsymbol{op}(v_1 : T_1, \dots, v_n : T_n)$ we check the proof obligation $\boldsymbol{SSOR_{op}} \asymp \boldsymbol{SSOR_{t_{op}^{nr}}}$.*

To check $\mathbf{SSOR_{op}} \asymp \mathbf{SSOR_{t_{op}^{nr}}}$ we have to check that $\mathbf{SSOR_{op}}$ and $\mathbf{SSOR_{t_{op}^{nr}}}$ are both sound and deterministic. It is enough to check only that $\mathbf{SSOR_{op}}$ is sound and deterministic, which we check explicitly using the corresponding definitions. We know that $\mathbf{SSOR_{t_{op}^{nr}}}$ is also sound and deterministic by construction. The rest of the properties of $\asymp$ we check explicitly using their definitions.

Lemma 42. *If*

- *the proof obligations from **Definition 70** hold* (1)
- *for each Σ if for each $exp_1, \dots, \exp_n$ $\boldsymbol{SSOR_{op^{nr}(\mathit{exp}_1,\dots,\exp_n)}}$ is proper then $\boldsymbol{SSOR_{t_{op}^{nr}}}$ is also proper on Σ* (2)

then $\boldsymbol{SSOR_{op}}$ corresponds to $\boldsymbol{t_{op}}$ and is proper.

Proof. Let us prove the desired property by induction over environment Σ. We have to prove that for each Σ if $\forall\ \Sigma'$ such that $\Sigma' < \Sigma$, $\mathbf{t_{op}} \overset{\Sigma'}{=} \mathbf{SSOR_{op}}$ and $\mathbf{SSOR_{op}}$ proper for Σ' then $\mathbf{t_{op}} \overset{\Sigma}{=} \mathbf{SSOR_{op}}$ and $\mathbf{SSOR_{op}}$ proper for Σ. Since the proof obligations hold (1) we can infer from **Lemma 24** that $\mathbf{SSOR_{op}} = \mathbf{SSOR_{t_{op}^{nr}}}$. By applying **Lemma 41** we get $\mathbf{t_{op}} \overset{\Sigma}{=} \mathbf{SSOR_{op}}$.

Let us prove that $\mathbf{t_{op}}$ proper for Σ. By applying **Lemma 40** and to the induction hypothesis we get that for each Σ if for each $exp_1, \dots, \exp_n$ $\mathbf{SSOR_{op^{nr}(\mathit{exp}_1,\dots,\exp_n)}}$ is proper. Therefore from (2) we get that $\mathbf{SSOR_{t_{op}^{nr}}}$ is proper on Σ. Since we already know that $\mathbf{SSOR_{op}} = \mathbf{SSOR_{t_{op}^{nr}}}$ we cam conclude that $\mathbf{SSOR_{op}}$ is proper on Σ.

□

Definition 71 (**SSOR** for a call of a composite recursive operation)**.** *We define an **SSOR** for a call $\boldsymbol{op}(exp_1, \dots, \exp_n)$ of a composite nonrecursive operation **op** as $\boldsymbol{SSOR_{op}}[v_1 := exp_1, \dots, v_n := exp_n]$. We denote it as $\boldsymbol{SSOR_{op(\mathit{exp}_1,\dots,\exp_n)}}$.*

Lemma 43. *If*

- *the proof obligations from **Definition 70** hold* (1)
- *for each* Σ *if for each* $exp_1, \ldots, \exp_n$ $\boldsymbol{SSOR}_{\boldsymbol{op}^{nr}(exp_1,\ldots,\exp_n)}$ *is proper then* $\boldsymbol{SSOR}_{t^{nr}_{op}}$ *is also proper on* Σ (2)

then $\boldsymbol{SSOR}_{\boldsymbol{op}(exp_1,\ldots,\exp_n)}$ *corresponds to* $\boldsymbol{op}(exp_1, \ldots, \exp_n)$ *and proper.*

Proof. From **Lemma 42** we know that $\mathbf{SSOR_{op}}$ corresponds to $\mathbf{t_{op}}$ and is proper. The rest of the proof is the same as for the basic operation (see **Lemma 35** and **Lemma 36**). □

Definition 43 is used by **Theorem 30**. The property (2) from **Definition 43** is inferred from the induction hypothesis of **Theorem 30**.

6.4.3 `skip` operator

Definition 72 (**SSOR** for `skip`)**.** *We define an **SSOR** for `skip` as:*

- *the pre-condition is* T*.*
- *the list of transformation rules is empty.*
- *the only frame rule is* $\langle \mathbb{U}, * \rangle$*.*

We denote it as $\boldsymbol{SSOR}_{\boldsymbol{skip}}$*.*

Lemma 44. $\boldsymbol{SSOR}_{\boldsymbol{skip}}$ *is proper.*

Proof.

- **Consistency:** The consistency is obvious since there is only one frame rule and no transformation rules.
- **Determinism:** The **SSOR** is deterministic since the frame rule covers the whole universe.
- **System invariant preservation:** Since the post-condition states that $\mathbf{sOld} = \mathbf{sNew}$ there is only one value of **sNew** which satisfies the post-condition, namely **sOld**. On the other hand, $\mathbf{pre} \wedge \mathbf{SysInv}[\mathbf{sOld}]$ any $\mathbf{sOld} = \mathbf{sNew}$ imply $\mathbf{SysInv}[\mathbf{sNew}]$. From this we can conclude that the **SSOR** preserves the system invariant.

□

Lemma 45. $\boldsymbol{SSOR}_{\boldsymbol{skip}}$ *corresponds to* `skip`*.*

Proof. For each $\mathbf{s}$, $\mathbf{s}'$, Σ $\Sigma \models \mathbf{s} \overset{\mathtt{skip}}{\rightarrow} \mathbf{s}' \Leftrightarrow \Sigma \models \mathbf{SysInvs} \wedge \mathbf{s} = \mathbf{s}' \Leftrightarrow \Sigma \models \mathbf{SysInvs} \wedge \mathbf{pre} \wedge \mathbf{post} \Leftrightarrow \Sigma \models \mathbf{s} \overset{\mathbf{SSOR_{skip}}}{\rightarrow} \mathbf{s}'$ where $[\![\mathbf{SSOR_{skip}}]\!] = \langle \mathbf{pre}, \mathbf{post} \rangle$.
□

6.4.4 Conditional statement

Definition 73 (**SSOR** for a conditional statement)**.** *We define an* ***SSOR*** *for* $\texttt{\textit{if}}\ (\varphi)\ \boldsymbol{t}_1\ \texttt{\textit{else}}\ \boldsymbol{t}_2$ *in the following way:*

- *the pre-condition is* $(\varphi \wedge \boldsymbol{pre}_1) \vee (\neg\varphi \wedge \boldsymbol{pre}_2)$.
- *the list of transformation rules is*
$\langle \varphi \wedge \psi^-_{S^t_{1,1}}, it^t_{1,1}, val_{1,1}\rangle, \ldots, \langle \varphi \wedge \psi^-_{S^t_{n^1_t,1}}, it^t_{n^1_t,1}, val_{n^1_t,1}\rangle,$
$\langle \neg\varphi \wedge \psi^-_{S^t_{1,2}}, it^t_{1,2}, val_{1,2}\rangle, \ldots, \langle \neg\varphi \wedge \psi^-_{S^t_{n^2_t,2}}, it^t_{n^2_t,2}, val_{n^2_t,2}\rangle.$
- *the list of frame rules is*
$\langle \varphi \wedge \psi^-_{S^f_{1,1}}, it^f_{1,1}\rangle, \ldots, \langle \varphi \wedge \psi^-_{S^f_{n^1_f,1}}, it^f_{n^1_f,1}\rangle,$
$\langle \neg\varphi \wedge \psi^-_{S^f_{1,2}}, it^f_{1,2}\rangle, \ldots, \langle \neg\varphi \wedge \psi^-_{S^f_{n^2_f,2}}, it^f_{n^2_f,2}\rangle.$

where $\boldsymbol{SSOR}_{\boldsymbol{t}_1}$ *has the following form:*

- *pre-condition* $\boldsymbol{pre}_1$.
- *transformation rules are* $\langle \psi^-_{S^t_{1,1}}, it^t_{1,1}, val_{1,1}\rangle, \ldots, \langle \psi^-_{S^t_{n^1_t,1}}, it^t_{n^1_t,1}, val_{n^1_t,1}\rangle$.
- *frame rules are* $\langle \psi^-_{S^f_{1,1}}, it^f_{1,1}\rangle, \ldots, \langle \psi^-_{S^f_{n^1_f,1}}, it^f_{n^1_f,1}\rangle$.

and $\boldsymbol{SSOR}_{\boldsymbol{t}_1}$ *has the following form:*

- *pre-condition* $\boldsymbol{pre}_2$.
- *transformation rules are* $\langle \psi^-_{S^t_{1,2}}, it^t_{1,2}, val_{1,2}\rangle, \ldots, \langle \psi^-_{S^t_{n^2_t,2}}, it^t_{n^2_t,2}, val_{n^2_t,2}\rangle$.
- *frame rules are* $\langle \psi^-_{S^f_{1,2}}, it^f_{1,2}\rangle, \ldots, \langle \psi^-_{S^f_{n^2_f,2}}, it^f_{n^2_f,2}\rangle$.

We denote it as $\boldsymbol{SSOR}_{\texttt{\textit{if}}\ (\varphi)\ \mathbf{t}_1\ \texttt{\textit{else}}\ \mathbf{t}_2}$.

Lemma 46. *if* $\boldsymbol{SSOR}_{\boldsymbol{t}_1}$ *and* $\boldsymbol{SSOR}_{\boldsymbol{t}_2}$ *are both proper and correspond to* $\boldsymbol{t}_1$ *and* $\boldsymbol{t}_2$*, respectively, then* $\boldsymbol{SSOR}_{\texttt{\textit{if}}\ (\varphi)\ \mathbf{t}_1\ \texttt{\textit{else}}\ \mathbf{t}_2}$ *is proper and corresponds to* $\texttt{\textit{if}}\ (\varphi)\ \boldsymbol{t}_1\ \texttt{\textit{else}}\ \boldsymbol{t}_2$.

Proof. If $\Sigma' \models \varphi$ holds, where $\Sigma' = \Sigma \cup \{\mathbf{sOld} \mapsto \mathbf{s}\}$, then both the term and the **SSOR** are equivalent to $\mathbf{t}_1$. In the other case, when $\Sigma' \models \neg\varphi$ holds, they both are equivalent to $\mathbf{t}_2$. Which implies that $\mathbf{SSOR}_{\texttt{if}\ (\varphi)\ \mathbf{t}_1\ \texttt{else}\ \mathbf{t}_2}$ corresponds to $\texttt{if}\ (\varphi)\ \mathbf{t}_1\ \texttt{else}\ \mathbf{t}_2$. In a similar way we can prove that since both $\mathbf{t}_1$ and $\mathbf{t}_2$ are proper then $\mathbf{SSOR}_{\texttt{if}\ (\varphi)\ \mathbf{t}_1\ \texttt{else}\ \mathbf{t}_2}$ is proper as well. □

6.4.5 Sequential composition

We can see that the operational semantics of sequential composition introduces an intermediate existentially quantified stereotype slice. Sequential composition is the only stereotype operation composer which has to deal not with two but with three stereotype slices. We denote this intermediate stereotype slice as **sTemp**.

In this subsection we consider sequential composition of $\mathbf{SSOR}_1$ and $\mathbf{SSOR}_2$. $\mathbf{SSOR}_1$ establishes the relation between stereotype slices **sOld** and **sTemp**. $\mathbf{SSOR}_2$ establishes the relation between stereotype slices **sTemp** and **sNew**. To simplify work with **sTemp** we introduce a special notation for the semantics of frame rules, transformation rules, and **SSOR**s for the first and the second parameters of sequential composition in the following way.

Definition 74 (Notation for the semantics of parameters of sequential composition)**.** *We denote the semantics of all elements relevant to the first and the second parameters of a sequential composition with subscripts 1 and 2, respectively.*

- $[\![\langle\psi_S^-, it, val\rangle]\!]_1 = \big(\forall \boldsymbol{v} : (\neg\psi_S^-) \vee \boldsymbol{sTemp.v}.it = val\big)$.
- $[\![\langle\psi_S^-, it\rangle]\!]_1 = \big(\forall \boldsymbol{v} : (\neg\psi_S^-) \vee \boldsymbol{sTemp.v}.it = \boldsymbol{sOld.v}.it\big)$.
- $[\![\langle\psi_S^-, it, val\rangle]\!]_2 = \big(\forall \boldsymbol{v} : (\neg\psi_S^-) \vee \boldsymbol{sNew.v}.it = val\big)$.
- $[\![\langle\psi_S^-, it\rangle]\!]_2 = \big(\forall \boldsymbol{v} : (\neg\psi_S^-) \vee \boldsymbol{sNew.v}.it = \boldsymbol{sTemp.v}.it\big)$.
- *We construct* $[\![\boldsymbol{SSOR}]\!]_1$ *and* $[\![\boldsymbol{SSOR}]\!]_2$ *from* $[\![\boldsymbol{SSOR}]\!]$ *by replacing all occurrences of* $[\![Q]\!]$ *by* $[\![Q]\!]_1$ *and* $[\![Q]\!]_2$ *respectively, where* Q *is either a frame or a transformation rule.*

The main challenge of constructing an **SSOR** of a sequential composition is the elimination of the intermediate stereotype slice **sTemp**. Below we provide auxiliary lemmas which are used to eliminate the intermediate slice from various parts of an **SSOR**.

Lemma 47 (Update of a formula by an **SSOR**)**.** *For each* Σ, $\boldsymbol{s}$, $\boldsymbol{s}'$, $\boldsymbol{s}''$, *and a proper* ***SSOR*** *such that* $\Sigma \models \boldsymbol{s} \overset{\boldsymbol{SSOR}}{\rightarrow} \boldsymbol{s}''$ *if all transformation and frame rules of the* ***SSOR*** *which affect a stereotype item "it" are* $\langle\psi_{S_1^t}^-, it, val_1\rangle, \ldots, \langle\psi_{S_n^t}^-, it, val_n\rangle$ *and* $\langle\psi_{S^f}^-, it\rangle$ *then, under the assumption that* $\Sigma' \models \boldsymbol{SysInv}[\boldsymbol{sOld}] \wedge \boldsymbol{pre}_{\boldsymbol{SSOR}}$ *holds, for each formula* $\forall x_1, \ldots, x_m : \varphi$ *which depends on* $\boldsymbol{sTemp}.exp.it$ *the following holds:*
$\Sigma' \models \forall x_1, \ldots, x_m : \varphi$ *is equivalent to* $\Sigma' \models \forall x_1, \ldots, x_m :$ $\big(\bigwedge_{i=1}^n (exp \in S_i^t \Rightarrow \varphi[\boldsymbol{sTemp}.exp.it := val_i])\big) \wedge (exp \in S^f \Rightarrow \varphi)$
where $\Sigma' = \Sigma \cup \{\boldsymbol{sOld} \mapsto \boldsymbol{s}, \boldsymbol{sTemp} \mapsto \boldsymbol{s}''\ \boldsymbol{sNew} \mapsto \boldsymbol{s}'\}$ *and* $x_1, \ldots, x_m$ *is a possibly empty list of variables.*

Proof. Since **SSOR** is proper it is also deterministic which implies that $\Sigma' \models \left(\bigvee_{i=1}^{n} exp \in S_i^t\right) \vee (exp \in S^f)$. There are two possibilities: either the value of **sTemp**.*exp.it* is changed by a transformation rule or it is preserved by the frame rule. The structure of the proof is the same for both cases. Let us consider the first case. $exp \in S_{i_0}^t$, where $i_0 \in [1..n]$. In this case $\Sigma' \models \forall x_1, \ldots, x_m : \left(\bigwedge_{i=1}^{n}(exp \in S_i^t \Rightarrow \varphi[\mathbf{sTemp}.exp.it := val_i])\right) \wedge (exp \in S^f \Rightarrow \varphi)$ is equivalent to $\Sigma' \models \forall x_1, \ldots, x_m : \varphi[\mathbf{sTemp}.exp.it := val_{i_0}]$. From $\Sigma \models \mathbf{s} \overset{\mathbf{SSOR}}{\rightarrow} \mathbf{s''}$ we can infer that the last statement is equivalent to $\Sigma' \models \forall x_1, \ldots, x_m : \varphi$, which concludes the proof. □

Definition 75 (Update of a pre-condition by an **SSOR**)**.** *We construct an updated version of a pre-condition* ***pre****, which is a universally quantified formula, by an* ***SSOR*** *in the following way. First, to avoid names clashes, we replace all occurrences of* ***sOld*** *by* ***sTemp****. Then we eliminate all subterms of the form* ***sTemp****.exp.it by applying the transformation from* ***Lemma 47****. We denote the resulting pre-condition as*
updatePre(***SSOR***, ***pre***)

Lemma 48. *For each* ***SSOR*** *and* ***pre updatePre***(***SSOR***, ***pre***) *always terminates and the resulting pre-condition doesn't depend on* ***sTemp****.*

Proof. The initial pre-condition contains a finite number of occurrences of terms like **sTemp**.*exp.it*. During each iteration of the procedure we eliminate one of them. For this reason the procedure eventually terminates and when this happens the resulting term doesn't contain **sTemp**. □

Lemma 49. *For each* ***pre***, Σ, ***s***, ***s'***, ***s''***, *and a proper* ***SSOR*** *such that* $\Sigma \models \boldsymbol{s} \overset{\boldsymbol{SSOR}}{\rightarrow} \boldsymbol{s''}$, *under the assumption that* $\Sigma'' \models \boldsymbol{SysInv}[\boldsymbol{sOld}] \wedge \boldsymbol{pre}_{\boldsymbol{SSOR}}$ *holds,* $\Sigma' \models \boldsymbol{pre}$ *holds if and only if* $\Sigma'' \models \boldsymbol{updatePre}(\boldsymbol{SSOR}, \boldsymbol{pre})$, *where* $\Sigma' = \Sigma \cup \{\boldsymbol{sOld} \mapsto \boldsymbol{s''}, \boldsymbol{sNew} \mapsto \boldsymbol{s'}\}$ *and* $\Sigma'' = \Sigma \cup \{\boldsymbol{sOld} \mapsto \boldsymbol{s}, \boldsymbol{sTemp} \mapsto \boldsymbol{s''}\ \boldsymbol{sNew} \mapsto \boldsymbol{s'}\}$.

Proof. $\Sigma' \models \mathbf{pre}$ holds if and only if $\Sigma'' \models \mathbf{pre}[\mathbf{sOld} := \mathbf{sTemp}]$. Then by applying **Lemma 47** for each subterm of the form **sTemp**.*exp.it* we get the desired property. □

Definition 76 (Elimination of a subterm **sTemp**.*exp.it* from a frame rule)**.** *We assume that all transformation and frame rules of* ***SSOR*** *which affect a stereotype item "it" are* $\langle \psi^-_{S_1^t}, it, val_1 \rangle, \ldots, \langle \psi^-_{S_n^t}, it, val_n \rangle$ *and* $\langle \psi^-_{S^f}, it \rangle$. *An updated by an* ***SSOR****, free from a subterm* ***sTemp****.exp.it, version of a frame rule* $\langle \psi^-_S, it' \rangle$ *is* $\langle \psi^-_{S'}, it' \rangle$ *where:*

$$\psi^-_{S'} = \left(\bigvee_{i=1}^{n}(exp \in S_i^t \wedge \psi^-_S[\boldsymbol{sTemp}.exp.it := val_i])\right) \vee (exp \in S^f \wedge \psi^-_S)$$

We denote $\langle \psi^-_{S'}, it' \rangle$ *as*
eliminateFromFRule(***sTemp****.exp.it*, $\langle \psi^-_{S'}, it' \rangle$, ***SSOR***).

Lemma 50. *For each $\langle\psi^-_S, it'\rangle$, Σ, $\boldsymbol{s}$, $\boldsymbol{s'}$, $\boldsymbol{s''}$, and a proper* ***SSOR*** *such that $\Sigma \models \boldsymbol{s} \overset{\boldsymbol{SSOR}}{\to} \boldsymbol{s''}$, under the assumption that $\Sigma' \models \boldsymbol{SysInv}[\boldsymbol{sOld}] \wedge \boldsymbol{pre}_{\boldsymbol{SSOR}}$ holds, $\Sigma' \models [\![\langle\psi^-_S, it'\rangle]\!]_2$ holds if and only if*
$\Sigma' \models [\![\boldsymbol{eliminateFromFRule}(\boldsymbol{sTemp}.exp.it, \langle\psi^-_{S'}, it'\rangle, \boldsymbol{SSOR})]\!]_2$ holds, where $\Sigma' = \Sigma \cup \{\boldsymbol{sOld} \mapsto \boldsymbol{s}, \boldsymbol{sTemp} \mapsto \boldsymbol{s''}\ \boldsymbol{sNew} \mapsto \boldsymbol{s'}\}$.

Proof.

$$\begin{array}{l}
\Sigma' \models [\![\langle\psi^-_S, it'\rangle]\!]_2 \Leftrightarrow \\
\Sigma' \models \forall \mathbf{v} : \psi^-_S \Rightarrow \mathbf{sNew}.\mathbf{v}.it' = \mathbf{sTemp}.\mathbf{v}.it' \Leftrightarrow \\
\Sigma' \cup \{\mathbf{sTemp'} \mapsto \mathbf{s''}\} \models \forall \mathbf{v} : \psi^-_S \Rightarrow \mathbf{sNew}.\mathbf{v}.it' = \mathbf{sTemp'}.\mathbf{v}.it' \overset{\textbf{Lemma 47}}{\Leftrightarrow} \\
\Sigma' \cup \{\mathbf{sTemp'} \mapsto \mathbf{s''}\} \models \forall \mathbf{v} : (exp \in S^f \Rightarrow (\psi^-_S \Rightarrow \mathbf{sNew}.\mathbf{v}.it' = \mathbf{sTemp'}.\mathbf{v}.it'))\wedge \\
\big(\bigwedge_{i=1}^n (exp \in S^t_i \Rightarrow (\psi^-_S \Rightarrow \mathbf{sNew}.\mathbf{v}.it' = \mathbf{sTemp'}.\mathbf{v}.it')[\mathbf{sTemp}.exp.it := val_i])\big) \Leftrightarrow \\
\Sigma' \cup \{\mathbf{sTemp'} \mapsto \mathbf{s''}\} \models \forall \mathbf{v} : ((exp \in S^f \wedge \psi^-_S) \Rightarrow \mathbf{sNew}.\mathbf{v}.it' = \mathbf{sTemp}.\mathbf{v}.it')\wedge \\
\big(\bigwedge_{i=1}^n ((exp \in S^t_i \wedge \psi^-_S[\mathbf{sTemp}.exp.it := val_i]) \Rightarrow \mathbf{sNew}.\mathbf{v}.it' = \mathbf{sTemp'}.\mathbf{v}.it')\big) \Leftrightarrow \\
\Sigma' \models \forall \mathbf{v} : ((exp \in S^f \wedge \psi^-_S) \Rightarrow \mathbf{sNew}.\mathbf{v}.it' = \mathbf{sTemp}.\mathbf{v}.it')\wedge \\
\big(\bigwedge_{i=1}^n ((exp \in S^t_i \wedge \psi^-_S[\mathbf{sTemp}.exp.it := val_i]) \Rightarrow \mathbf{sNew}.\mathbf{v}.it' = \mathbf{sTemp}.\mathbf{v}.it')\big) \Leftrightarrow \\
\Sigma' \models \forall \mathbf{v} : \big(\bigvee_{i=1}^n ((exp \in S^t_i \wedge \psi^-_S[\mathbf{sTemp}.exp.it := val_i])\big) \vee ((exp \in S^f \wedge \psi^-_S) \Rightarrow \\
\mathbf{sNew}.\mathbf{v}.it' = \mathbf{sTemp}.\mathbf{v}.it' \Leftrightarrow \\
\Sigma' \models [\![\mathbf{eliminateFromFRule}(\mathbf{sTemp}.exp.it, \langle\psi^-_{S'}, it'\rangle, \mathbf{SSOR})]\!]_2
\end{array}$$

□

Definition 77 (Update of a frame rule by an **SSOR**)**.** *We construct an updated version of a frame rule $\langle\psi^-_S, it'\rangle$ by an* ***SSOR*** *in the following way. First, to avoid name clashes, we replace all occurrences of* ***sOld*** *by* ***sTemp****. Then we eliminate all subterms of the form* ***sTemp****.exp.it by applying the* ***eliminateFromTRule*** *transformation. We denote the resulting frame rule as $\boldsymbol{updateFRule}(\boldsymbol{SSOR}, \langle\psi^-_S, it'\rangle)$*

Lemma 51. *For each $\langle\psi^-_S, it'\rangle$, Σ, $\boldsymbol{s}$, $\boldsymbol{s'}$, $\boldsymbol{s''}$, and a proper* ***SSOR*** *such that $\Sigma \models \boldsymbol{s} \overset{\boldsymbol{SSOR}}{\to} \boldsymbol{s''}$ and $\Sigma'' \models \boldsymbol{SysInv}[\boldsymbol{sOld}] \wedge \boldsymbol{pre}_{\boldsymbol{SSOR}}$ hold, $\Sigma' \models [\![\langle\psi^-_S, it'\rangle]\!]$ holds if and only if $\Sigma'' \models [\![\boldsymbol{updateFRule}(\boldsymbol{SSOR}, \langle\psi^-_S, it'\rangle)]\!]_2$, where $\Sigma' = \Sigma \cup \{\boldsymbol{sOld} \mapsto \boldsymbol{s''}, \boldsymbol{sNew} \mapsto \boldsymbol{s'}\}$ and $\Sigma'' = \Sigma \cup \{\boldsymbol{sOld} \mapsto \boldsymbol{s}, \boldsymbol{sTemp} \mapsto \boldsymbol{s''}\ \boldsymbol{sNew} \mapsto \boldsymbol{s'}\}$.*

Proof. $\Sigma' \models [\![\langle\psi^-_S, it'\rangle]\!]$ holds if and only if $\Sigma'' \models [\![\langle\psi^-_S[\mathbf{sOld} := \mathbf{sTemp}], it'\rangle]\!]$ holds. Then by applying **Lemma 50** for each subterm of the form **sTemp**.*exp.it* we get the desired property. □

Lemma 52. *For each* ***SSOR*** *and $\langle\psi^-_S, it'\rangle$ $\boldsymbol{updateFRule}(\boldsymbol{SSOR}, \langle\psi^-_S, it'\rangle)$ always terminates. The resulting frame rule doesn't depend on* ***sTemp****.*

Proof. The proof is the same as for **Lemma 48**. □

Definition 78 (Elimination of a subterm **sTemp**.*exp.it* from a transformation rule)**.** *We assume that all transformation and frame rules of an* ***SSOR*** *which affect a stereotype item "it" are $\langle\psi^-_{S^t_1}, it, val_1\rangle, \ldots, \langle\psi^-_{S^t_n}, it, val_n\rangle$ and $\langle\psi^-_{S^f}, it\rangle$. An updated by an* ***SSOR****, free from a subterm* ***sTemp****.exp.it,* *version of a transformation rule $\langle \psi^-_S, it', val\rangle$ is the following list of transformation rules $\langle \psi^-_{S'_0}, it', val_0\rangle, \ldots, \langle \psi^-_{S'_n}, it', val_n\rangle$ where:*

- $val_0 = val$
- $\psi^-_{S'_0} = exp \in S^f \wedge \psi^-_S$
- $val_i = val[\boldsymbol{sTemp}.exp.it := val_i]$ *where* $i \in [1..n]$
- $\psi^-_{S'_i} = exp \in S^t_i \wedge \psi^-_S[\boldsymbol{sTemp}.exp.it := val_i]$ *where* $i \in [1..n]$

We denote $\langle \psi^-_{S'_0}, it', val_0\rangle, \ldots, \langle \psi^-_{S'_n}, it', val_n\rangle$ *as*
$\boldsymbol{eliminateFromTRule}(\boldsymbol{sTemp}.exp.it, \langle \psi^-_{S'}, it', val\rangle, \boldsymbol{SSOR})$.

Lemma 53. *For each* $\langle \psi^-_S, it', val\rangle$, Σ, $\boldsymbol{s}$, $\boldsymbol{s'}$, $\boldsymbol{s''}$, *and a proper* $\boldsymbol{SSOR}$ *such that* $\Sigma \models \boldsymbol{s} \overset{\boldsymbol{SSOR}}{\rightarrow} \boldsymbol{s''}$ *,under the assumption that* $\Sigma' \models \boldsymbol{SysInv}[\boldsymbol{sOld}] \wedge \boldsymbol{pre}_{\boldsymbol{SSOR}}$ *holds,* $\Sigma' \models [\![\langle \psi^-_S, it', val\rangle]\!]_2$ *holds if and only if*
$\Sigma' \models \bigwedge_{i=0}^n [\![\langle \psi^-_{S'_i}, it', val_i\rangle]\!]_2$ *holds, where* $\langle \psi^-_{S'_0}, it', val_0\rangle, \ldots, \langle \psi^-_{S'_n}, it', val_n\rangle$ *is equal to* $\boldsymbol{eliminateFromTRule}(\boldsymbol{sTemp}.exp.it, \langle \psi^-_{S'}, it', val\rangle, \boldsymbol{SSOR})$ *and* $\Sigma' = \Sigma \cup \{\boldsymbol{sOld} \mapsto \boldsymbol{s}, \boldsymbol{sTemp} \mapsto \boldsymbol{s''}\ \boldsymbol{sNew} \mapsto \boldsymbol{s'}\}$.

Proof.

$$\begin{aligned}
&\Sigma' \models [\![\langle \psi^-_S, it', val\rangle]\!]_2 \Leftrightarrow \\
&\Sigma' \models \forall \mathbf{v} : \psi^-_S \Rightarrow \mathbf{sNew}.\mathbf{v}.it' = val \overset{\textbf{Lemma 47}}{\Leftrightarrow} \\
&\Sigma' \models \forall \mathbf{v} : (exp \in S^f \Rightarrow (\psi^-_S \Rightarrow \mathbf{sNew}.\mathbf{v}.it' = val)) \wedge \\
&\big(\textstyle\bigwedge_{i=1}^n (exp \in S^t_i \Rightarrow (\psi^-_S \Rightarrow \mathbf{sNew}.\mathbf{v}.it' = val)[\mathbf{sTemp}.exp.it := val_i])\big) \Leftrightarrow \\
&\Sigma' \models \forall \mathbf{v} : \Big(\textstyle\bigwedge_{i=0}^n (\psi^-_{S'_i} \Rightarrow \mathbf{sNew}.\mathbf{v}.it' = val_i)\Big) \Leftrightarrow \\
&\Sigma' \models \textstyle\bigwedge_{i=0}^n \Big(\forall \mathbf{v} : (\psi^-_{S'_i} \Rightarrow \mathbf{sNew}.\mathbf{v}.it' = val_i)\Big) \Leftrightarrow \\
&\Sigma' \models \textstyle\bigwedge_{i=0}^n [\![\langle \psi^-_{S'_i}, it', val_i\rangle]\!]_2
\end{aligned}$$

□

Definition 79 (Update of a list of transformation rules by an **SSOR**)**.** *We construct an updated version of a list of transformation rules* $\langle \psi^-_{S_1}, it_1, val_1\rangle$, $\ldots, \langle \psi^-_{S_n}, it_n, val_n\rangle$ *by an* $\boldsymbol{SSOR}$ *in the following way. First, to avoid name clashes, we replace all occurrences of* $\boldsymbol{sOld}$ *by* $\boldsymbol{sTemp}$*. Then we apply the following iterative procedure. We choose a transformation rule which contains the biggest number of subterms of form* $\boldsymbol{sTemp}.exp.it$*. If there are several such rules we choose any of them. Let us denote the index of the chosen rule as* i_0*. Then we choose any subterm of the rule* i_0 *that has form* $\boldsymbol{sTemp}.exp.it$*. We remove the chosen rule from the list of the transformation rules and instead we add the list of rules*
$\boldsymbol{eliminateFromTRule}(\boldsymbol{sTemp}.exp.it, \langle \psi^-_{S'}, it', val\rangle, \boldsymbol{SSOR})$*. We denote the resulting list of transformation rules as*
$\boldsymbol{updateTRule}(\boldsymbol{SSOR}, \langle \psi^-_{S_1}, it_1, val_1\rangle, \ldots, \langle \psi^-_{S_n}, it_n, val_n\rangle)$.

Lemma 54. *For each* $\boldsymbol{SSOR}$ *and list of transformation rules* $\langle \psi^-_{S_1}, it_1, val_1\rangle, \dots, \langle \psi^-_{S_n}, it_n, val_n\rangle$ $\boldsymbol{updateTRule}(\boldsymbol{SSOR}, \langle \psi^-_{S_1}, it_1, val_1\rangle, \dots, \langle \psi^-_{S_n}, it_n, val_n\rangle)$ *always terminates and the resulting list of transformation rules doesn't depend on* $\boldsymbol{sTemp}$.

Proof. We use the lexicographic order on pairs of natural numbers as a measure function to guarantee the termination of the procedure. The first number is the maximum of number occurrences of subterm of the form **sTemp**.*exp.it* over all transformation rules which participate in the iteration. The second one is the number of transformation rules which has exactly this number of such subterms. If the second number is greater then 1 then we decrement it during the operation execution. Otherwise, if it is equal to 1, then the iteration execution decrements the first number. In the later case the value of the second number can increase, but it does not matter because according to the definition of the lexicographic order the resulting pair of numbers is strictly less then the initial one. The procedure terminates when the first number is equal to 0, which implies that the output of the procedure doesn't depend on **sTemp**. □

Lemma 55. *For each* $\langle \psi^-_{S_1}, it_1, val_1\rangle, \dots, \langle \psi^-_{S_n}, it_n, val_n\rangle$, Σ, $\boldsymbol{s}$, $\boldsymbol{s}'$, $\boldsymbol{s}''$, *and a proper* $\boldsymbol{SSOR}$ *such that* $\Sigma \models \boldsymbol{s} \overset{\boldsymbol{SSOR}}{\rightarrow} \boldsymbol{s}''$, *under the assumption that* $\Sigma'' \models \boldsymbol{SysInv}[\boldsymbol{sOld}] \wedge \boldsymbol{pre}_{\boldsymbol{SSOR}}$ *holds,* $\Sigma' \models \bigwedge_{i=1}^{n} [\![\langle \psi^-_{S_i}, it_i, val_i\rangle]\!]$ *holds if and only if* $\Sigma' \models \bigwedge_{i=1}^{n'} [\![\langle \psi^-_{S'_i}, it'_i, val'_i\rangle]\!]_2$, *where* $\langle \psi^-_{S'_1}, it'_1, val'_1\rangle, \dots, \langle \psi^-_{S'_{n'}}, it'_{n'}, val'_{n'}\rangle$ *is equal to* $\boldsymbol{updateTRule}(\boldsymbol{SSOR}, \langle \psi^-_{S_1}, it_1, val_1\rangle, \dots, \langle \psi^-_{S_n}, it_n, val_n\rangle)$, $\Sigma' = \Sigma \cup \{\boldsymbol{sOld} \mapsto \boldsymbol{s}'', \boldsymbol{sNew} \mapsto \boldsymbol{s}'\}$, *and* $\Sigma'' = \Sigma \cup \{\boldsymbol{sOld} \mapsto \boldsymbol{s}, \boldsymbol{sTemp} \mapsto \boldsymbol{s}''\ \boldsymbol{sNew} \mapsto \boldsymbol{s}'\}$.

Proof. $\Sigma' \models \bigwedge_{i=1}^{n} [\![\langle \psi^-_{S_i}, it_i, val_i\rangle]\!]$ holds if and only if $\Sigma' \models \bigwedge_{i=1}^{n} [\![\langle \psi^-_{S_i}, it_i, val_i\rangle[\mathbf{sOld} := \mathbf{sTemp}]]\!]$ holds. Then by applying **Lemma 53** for each subterm of the form **sTemp**.*exp.it* we get the desired property. □

Definition 80 (Update of an **SSOR** by another **SSOR**)**.** *We construct an updated version of an* $\boldsymbol{SSOR}_2$ *by an* $\boldsymbol{SSOR}_1$ *in the following way:*

- *pre-condition:*
  $\boldsymbol{updatePre}(\boldsymbol{SSOR}_1, \boldsymbol{pre})$.
- *transformation rules:*
  $\boldsymbol{updateTRule}(\boldsymbol{SSOR}_1, \langle \psi^-_{S^t_1}, it^t_1, val_1\rangle, \dots, \langle \psi^-_{S^t_{n_t}}, it^t_{n_t}, val_{n_t}\rangle)$.
- *frame rules:*
  $\boldsymbol{updateFRule}(\boldsymbol{SSOR}_1, \langle \psi^-_{S^f_1}, it^f_1\rangle), \dots,$
  $\boldsymbol{updateFRule}(\boldsymbol{SSOR}_2, \langle \psi^-_{S^f_{n_f}}, it^f_{n_f}\rangle)$.

where $\boldsymbol{SSOR}_2$ *is:*

- *pre-condition:* $\boldsymbol{pre}$.
- *transformation rules:* $\langle \psi^-_{S^t_1}, it^t_1, val_1\rangle, \ldots, \langle \psi^-_{S^t_{n_t}}, it_{n_t}, val^t_{n_t}\rangle$.
- *frame rules:* $\langle \psi^-_{S^f_1}, it^f_1\rangle, \ldots, \langle \psi^-_{S^f_{n_f}}, it^f_{n_f}\rangle$.

We denote the resulting $\boldsymbol{SSOR}$ *as* $\boldsymbol{updateSSOR}(\boldsymbol{SSOR}_1, \boldsymbol{SSOR}_2)$.

Definition 81 (**SSOR** for a sequential composition)**.** *We define the pre-condition of* $\boldsymbol{SSOR}_{\mathbf{t}_1;\mathbf{t}_2}$ *as* $\boldsymbol{pre}_1 \wedge \boldsymbol{updatePre}(\boldsymbol{SSOR}_1, \boldsymbol{pre}_2)$, *where* $\boldsymbol{pre}_1$ *and* $\boldsymbol{pre}_2$ *are pre-conditions of* $\boldsymbol{SSOR}_{\mathbf{t}_1}$ *and* $\boldsymbol{SSOR}_{\mathbf{t}_2}$, *respectively. Let us now define the transformation and frame rules of* $\boldsymbol{SSOR}_{\mathbf{t}_1;\mathbf{t}_2}$ *for each stereotype item "it". We assume that all transformation and frame rules of* $\boldsymbol{SSOR}_{\mathbf{t}_1}$, *which affect the stereotype item "it", are* $\langle \psi^-_{S^t_{1,1}}, it, val_{1,1}\rangle, \ldots, \langle \psi^-_{S^t_{n_1,1}}, it, val_{n_1,1}\rangle$ *and* $\langle \psi^-_{S^f_1}, it\rangle$. *We also assume that all transformation and frame rules of* $\boldsymbol{updateSSOR}(\boldsymbol{SSOR}_{\mathbf{t}_1}, \boldsymbol{SSOR}_{\mathbf{t}_2})$, *which affect a stereotype item "it", are* $\langle \psi^-_{S^t_{1,2}}, it, val_{1,2}\rangle, \ldots, \langle \psi^-_{S^t_{n_2,2}}, it, val_{n_2,2}\rangle$ *and* $\langle \psi^-_{S^f_2}, it\rangle$. *We define the transformation rules of the resulting* $\boldsymbol{SSOR}$ *as* $\langle \psi^-_{S^t_{1,1}\cap S^f_2}, it, val_{1,1}\rangle, \ldots, \langle \psi^-_{S^t_{n_1,1}\cap S^f_2}, it, val_{n_1,1}\rangle, \langle \psi^-_{S^t_{1,2}}, it, val_{1,2}\rangle, \ldots, \langle \psi^-_{S^t_{n_2,2}}, it, val_{n_2,2}\rangle$. *We define the frame rules of the resulting* $\boldsymbol{SSOR}$ *as* $\langle \psi^-_{S^f_1\cap S^f_2}, it\rangle$.

Lemma 56. *If* $\boldsymbol{SSOR}_{\mathbf{t}_1}$ *corresponds to* $\mathbf{t}_1$ *and* $\boldsymbol{SSOR}_{\mathbf{t}_2}$ *corresponds to* $\mathbf{t}_2$ *then* $\boldsymbol{SSOR}_{\mathbf{t}_1;\mathbf{t}_2}$ *corresponds to* $\mathbf{t}_1;\mathbf{t}_2$.

Proof. To prove the correspondence it is enough to prove equivalence of pre-conditions, transformation rules, and frame rules.

The equivalence of the pre-conditions follows immediately from the definitions of the operational semantics of the sequential composition, the definition of $\mathbf{SSOR}_{\mathbf{t}_1;\mathbf{t}_2}$, and **Lemma 49**.

To prove the equivalence of the transformation rules it is enough to consider two cases for each affected object: the object is affected by the second term (and possibly by the first term also), the object is affected by the first term only. In the first case we can infer from **Lemma 55** that we can directly use the transformation rules of $\mathbf{updateSSOR}(\mathbf{SSOR}_1, \mathbf{SSOR}_2)$, which corresponds to the second part of the definition of transformation rules of $\mathbf{SSOR}_{\mathbf{t}_1;\mathbf{t}_2}$. The second case applies when the value of "*it*" of an object is affected by $\mathbf{t}_1$ and preserved by $\mathbf{t}_2$. To formalize this transformation it is enough to restrict a domain of transformation rules of $\mathbf{t}_1$ by objects which belong to an updated frame rule of $\mathbf{t}_2$, which corresponds to the first part of the definition of transformation rules of $\mathbf{SSOR}_{\mathbf{t}_1;\mathbf{t}_2}$. In this case we can infer equivalence form **Lemma 51**.

A value of "*it*" of an object is preserved by $\mathbf{SSOR_{t_1;t_2}}$ if and only if it is preserved by both $\mathbf{t}_1$ and $\mathbf{t}_2$. That is why we construct the domain of the frame rule of the resulting **SSOR** as intersection of the domains of the corresponding frame rules of $\mathbf{t}_1$ and $\mathbf{t}_1$. The equivalence follows from **Lemma 51**. □

Lemma 57. *If* $\boldsymbol{SSOR_{t_1}}$ *and* $\boldsymbol{SSOR_{t_2}}$ *are both proper then* $\boldsymbol{SSOR_{t_1;t_2}}$ *is also proper.*

Proof. First of all let us prove that the resulting **SSOR** is sound and deterministic. Let us consider $\mathbf{updateSSOR}(\mathbf{SSOR}_1, \mathbf{SSOR}_2)$. We construct it from if $\mathbf{SSOR_{t_2}}$ by applying the transformation from **Lemma 47** to preconditions, transformation, and frame rules. Since $\mathbf{SSOR_{t_1}}$ is sound and deterministic, the application of the transformation preserves soundness and determinism. By this we can conclude that $\mathbf{updateSSOR}(\mathbf{SSOR}_1, \mathbf{SSOR}_2)$ is also sound and deterministic. Let us now consider the sequential composition of $\mathbf{SSOR}_1$ and $\mathbf{updateSSOR}(\mathbf{SSOR}_1, \mathbf{SSOR}_2)$. We can see that we construct the resulting **SSOR** by partitioning all objects in the following regions: objects affected by $\mathbf{t}_2$ (and possibly by $\mathbf{t}_1$), objects affected by $\mathbf{t}_1$ only, and objects preserved by both $\mathbf{t}_1$ and $\mathbf{t}_2$. Since both $\mathbf{SSOR}_1$ and $\mathbf{updateSSOR}(\mathbf{SSOR}_1, \mathbf{SSOR}_2)$ are sound and deterministic and those three regions cover all objects and disjoint the resulting, **SSOR** is also sound and deterministic.

Let us now prove the system invariant preservation. Since $\mathbf{t}_1$ preserves the system invariant we can infer that $\mathbf{t}_1$ transforms a stereotype slice which preserves the system invariant into another slice which preserves the system invariant. The same is true for $\mathbf{t}_2$. By combining these two observations we get $\mathbf{t}_1; \mathbf{t}_2$ also preserves the system invariant. Since $\mathbf{SSOR_{t_1;t_2}}$ corresponds to $\mathbf{t}_1; \mathbf{t}_2$ it also preserves the system invariant. □

By this we complete the description of the sequential composition. The generation of an **SSOR** for a sequential composition is completely automatic and doesn't require any proof obligations.

6.4.6 Parallel composition

Definition 82 (**SSOR** for a parallel composition)**.** *We define an* ***SSOR*** *for* $\overset{f_{\boldsymbol{Inst}}}{\underset{i \in Ind}{\parallel}} \boldsymbol{op}(exp_1[i], \ldots, exp_n[i])$ *in the following way:*

- *pre-condition* $(\forall i \in Ind : \boldsymbol{pre}[i]) \wedge (\forall i \in Ind, j \in Ind \setminus \{j\} : \boldsymbol{Inst_{op}}[i] \sharp \boldsymbol{Inst_{op}}[j]) \wedge \overset{f_{\boldsymbol{Inst}}}{\underset{i \in Ind}{\bigcup}} \boldsymbol{Inst_{op}}[i] = \boldsymbol{Inst}$,

  where $\boldsymbol{Inst} = \overset{f_{\boldsymbol{Inst}}}{\underset{i \in Ind}{\bigcup}} \boldsymbol{Inst_{op}}[i]$.

- *transformation rules are* $\langle \exists i \in Ind : \psi^{-}_{S^t_1[i]}, it^t_1, val_1[i]\rangle, \ldots, \langle \exists i \in Ind : \psi^{-}_{S^t_{n_t}[i]}, it^t_{n_t}, val_{n_t}[i]\rangle$.
- *frame rules are* $\langle \exists i \in Ind : \psi^{-}_{S^f_1[i] \cap S_{\boldsymbol{Inst}}[i]}, it^f_1\rangle, \ldots,$ $\langle \exists i \in Ind : \psi^{-}_{S^f_{n_f}[i] \cap S_{\boldsymbol{Inst}}[i]}, it^f_{n_f}\rangle$ *and* $\langle \overline{\boldsymbol{InstEl}(\boldsymbol{Inst})}, *\rangle$.

where $\boldsymbol{SSOR}_{\boldsymbol{op}(exp_1[i],\ldots,exp_n[i])}$ *has following form:*

- *pre-condition* $\boldsymbol{pre}[i]$.
- *transformation rules are* $\langle \psi^{-}_{S^t_1[i]}, it^t_1, val_1[i]\rangle, \ldots, \langle \psi^{-}_{S^t_{n_t}[i]}, it^t_{n_t}, val_{n_t}[i]\rangle$.
- *frame rules are* $\langle \psi^{-}_{S^f_1[i]}, it^f_1\rangle, \ldots, \langle \psi^{-}_{S^f_{n_f}[i]}, it^f_{n_f}\rangle$.

and for each $i \in Ind$ $S_{\boldsymbol{Inst}}[i] = \boldsymbol{InstEl}(\boldsymbol{Inst_{op}}[i])$.
We denote it as $\boldsymbol{SSOR}_{\overset{f_{\boldsymbol{Inst}}}{\underset{i \in Ind}{\parallel}} \boldsymbol{op}(exp_1[i],\ldots,exp_n[i])}$.

Lemma 58. *If for each i from Ind* $\boldsymbol{SSOR}_{\boldsymbol{op}(exp_1[i],\ldots,exp_n[i])}$ *correspond to* $\boldsymbol{op}(exp_1[i], \ldots, exp_n[i])$ *then* $\boldsymbol{SSOR}_{\overset{f_{\boldsymbol{Inst}}}{\underset{i \in Ind}{\parallel}} \boldsymbol{op}(exp_1[i],\ldots,exp_n[i])}$ *corresponds to* $\overset{f_{\boldsymbol{Inst}}}{\underset{i \in Ind}{\parallel}} \boldsymbol{op}(exp_1[i], \ldots, exp_n[i])$.

Proof. Let us consider the different parts of **SSOR** and check that their behavior corresponds to the operational semantics:

- **Pre-condition:** For each initial state which satisfies the system invariant, the operational semantics progresses if and only if the pre-conditions of all operation calls are satisfied, $\mathbf{Inst_{op}}[i]$ are disjoint for different i, and **Inst** is equal to the union of all $\mathbf{Inst_{op}}[i]$. The last two properties are explicitly represented by the pre-condition. We quantify pre-conditions of different operation calls over elements of Ind to represent the first property.

- **Transformation rules:** Since we know that the operations affect disjoint stereotype instances we can be sure that there is no interference between different operation calls. That is why we construct the list of transformation rules of the resulting **SSOR** by merging the lists of transformations for all operation calls. We do it by adding an extra quantifier to each transformation rule. From a logical perspective the quantifier creates a copy of a transformation rule for each i from Ind.

- **Frame rules:** There are two sources of frame rules in the operational semantics: an explicit rule $\langle\overline{\mathbf{InstEl}(\mathbf{Inst})}, *\rangle$, and frame rules of restricted operation calls. The first one we explicitly add to the list of frame rules of the resulting **SSOR**. The frame rules of restricted operation calls differ from the frame rules of non-restricted operations by limiting them on $\mathbf{InstEl}(\mathbf{Inst_{op}}[i])$. We construct a list of frame rules which corresponds to the restricted operations call by using quantification over Ind , the same way as we did it for the transformation rules, and by intersecting their domains with $\mathbf{InstEl}(\mathbf{Inst_{op}}[i])$.

□

Lemma 59. *If for each i from Ind, the $\boldsymbol{SSOR}_{\boldsymbol{op}(exp_1[i],\ldots,exp_n[i])}$ are proper then $\boldsymbol{SSOR}_{f_{\boldsymbol{Inst}} \underset{i\in Ind}{\|} \boldsymbol{op}(exp_1[i],\ldots,exp_n[i])}$ is also proper.*

Proof.
- **Soundness:** Since the $\mathbf{SSOR}_{\mathbf{op}(exp_1[i],\ldots,exp_n[i])}$ are proper, there is no inconsistency between two elements of the same operation call. On the other hand it is possible that two elements of different calls are inconsistent, or an element of a call is inconsistent with $\langle\overline{\mathbf{InstEl}(Inst)}, *\rangle$. We know that transformation rules which correspond to $\mathbf{SSOR}_{\mathbf{op}(exp_1[i],\ldots,exp_n[i])}$ are limited by $\mathbf{InstEl}(\mathbf{Inst_{op}}[i])$. By construction we know that frame rules which correspond to $\mathbf{SSOR}_{\mathbf{op}(exp_1[i],\ldots,exp_n[i])}$ are also limited by $\mathbf{InstEl}(\mathbf{Inst_{op}}[i])$. By this we can conclude that there is no inconsistency between elements of different calls. On the other hand $\langle\overline{\mathbf{InstEl}(Inst)}, *\rangle$ is disjoint from $\mathbf{InstEl}(\mathbf{Inst_{op}}[i])$ for each i. By this we can conclude that there is no inconsistency between $\langle\overline{\mathbf{InstEl}(\mathbf{Inst})}, *\rangle$ and elements of different calls.

- **Determinism:** Since we know that for each i from Ind $\mathbf{SSOR}_{\mathbf{op}(exp_1[i],\ldots,exp_n[i])}$ is deterministic and the resulting **SSOR** behaves as $\mathbf{SSOR}_{\mathbf{op}(exp_1[i],\ldots,exp_n[i])}$, we can conclude that on $\bigcup_{i\in Ind} \mathbf{InstEl}(\mathbf{Inst_{op}}[i])$ the resulting **SSOR** is deterministic. This implies that on $\mathbf{InstEl}(\mathbf{Inst})$ the resulting **SSOR** is deterministic. On the other hand, the frame rule $\langle\overline{\mathbf{InstEl}(\mathbf{Inst})}, *\rangle$ defines the resulting **SSOR** in a deterministic way on on $\overline{\mathbf{InstEl}(\mathbf{Inst})}$.

- **Invariant preservation:** For each i from Ind preservation of the system invariant on all instances from $\mathbf{Inst_{op}}[i]$ follows from the system invariant preservation by $\mathbf{SSOR}_{\mathbf{op}(exp_1[i],\ldots,exp_n[i])}$. This implies the system invariant preservation by all instances from **Inst**. On the other hand, all instances outside of **Inst** are obviously preserved by the frame rule $\langle\overline{\mathbf{InstEl}(\mathbf{Inst})}, *\rangle$.

□

6.5 Universal transformation

Now we have enough machinery to construct universal transformations. A universal transformation of a stereotype *St* should be expressive enough to formalize any computable update of a stereotype slice of the stereotype *St*. Here we assume that a stereotype characterizes some binary relation. This binary relation represents a semantic relation or a heap reference between objects. From this perspective a stereotype slice is a graph whose nodes are allocated objects and edges are semantic relations or heap references. Stereotype instances correspond to connected components of the graph.

Any update of a graph can be represented as a sequence of additions and removals of edges. Because of commutativity of addition and removal of edges we can rearrange them to have all removals of edges before the additions. Then we can split all removals on groups in such a way that each group affects exactly one stereotype instance from the initial stereotype slice and different groups affects different stereotype instances. Each of the groups of removals splits an instance on several instances. We call a stereotype operation which is equivalent to such a group of removals universal removal and denote it as **removeSetStereotype**. Then we perform a similar reordering of additions. We can split all additions in groups in such a way that each group merges several stereotype instances into one stereotype instance, and different groups merge different stereotype instances. We call a stereotype operation which is equivalent to such a group of additions universal addition and denote it as **addSetStereotype**.

By putting together all transformations above, we can see that an arbitrary stereotype transformation can be represented as:

$$\left(\overset{f_{\mathbf{Inst}_1}}{\underset{i\in Ind_1}{\|}} \mathbf{removeSetStereotype}(\ldots)\right) ; \left(\overset{f_{\mathbf{Inst}_2}}{\underset{i\in Ind_2}{\|}} \mathbf{addSetStereotype}(\ldots)\right)$$

Here we omit the actual parameters of the operations because they can vary from one stereotype to another. The above explicit description of the universal transformation reduces the construction of the universal transformation for a stereotype to the construction of **removeSetStereotype** and **addSetStereotype** for the same stereotype.

In **Appendix A** and **Appendix B** we construct **removeSetStereotype** and **addSetStereotype** for the `Sequence` and `Tree` stereotypes. As stated above, by this we construct universal transformations for the `Sequence` and `Tree` stereotypes. In the similar way the universal transformation can be constructed for DAGs, graphs, and planar graphs. These stereotypes can be used to verify many of practical examples.

Unfortunately not for all stereotypes it is possible to construct stereotype operations **removeSetStereotype** and **addSetStereotype** in such a way that they preserve the system invariant. For instance, it is impossible to construct stereotype operations **removeSetStereotype** and **addSet-**

Stereotype for the cyclic list. Splitting a cyclic list into two involves two additions and removals of edges. Nevertheless, we cannot consider them in isolation. Each of the operations violates the system invariant, but applied together they preserve it.

If we cannot construct **removeSetStereotype** and **addSetStereotype** for a stereotype St we cannot construct the universal transformation for the stereotypeSt either. This problem can be partially addressed in the following way. We construct a stereotype St' with a weaker stereotype invariant. The invariant of St' has to be simple enough to enable the construction of **removeSetStereotype**, **addSetStereotype**, and the universal transformation for St'. St' can be used instead of St in the source code specifications. The omitted invariants of St can be specified as glue invariants. For instance in **Section 5.4** we use a sequence stereotype to describe a cyclic list.

Chapter 7

Related work

In this chapter we establish connections between the stereotype-based approach and other approaches to the specification and verification of heap structures and semantical relations. Below we propose a classification of existing approaches according to the methods which they use to verify logical properties:

- **Approaches based on decidable procedures.** This group includes all approaches which for any given verification problem eventually terminate. The group consists of the following sub-groups:
  - Type systems for alias control. Approaches from this sub-group use types or type annotations to specify heap properties. Type system rules check the validity of the provided annotations.
  - Approaches based on decidable procedures for a logic theory. Approaches from this sub-group use the weakest pre-condition calculus or a symbolic computation to reduce a given verification problem to the validation of a formula from a decidable logic.
  - Approaches based on abstract interpretation. Approaches from this group use a finite approximation of the possibly infinite set of the heap states. A fix-point computation is used to find an approximation of the possible heap states at any point of a given program. Later on these approximations are used to verify the validity of the heap specifications of the program.
- **Approaches based on an automatic theorem prover.** Here we include all the approaches which use **SMT** and tableaux based **FOL** theorem provers. The main issue for the tools from this group is how to verify the properties which use transitive closure and inductive definitions in **FOL**. On one hand these properties are the only way to specify most of the heap properties, on the other hand the verification of such properties requires inductive proofs which cannot be expressed

in **FOL**. We classify the approaches from this group according to the way in which they address this issue:

- Approaches based on the approximation of induction in **FOL**. These approaches choose a finite set of **FOL** formulas (whose proofs require induction) and use them as axioms. The approximation is sound but not complete. Any valid formula can be verified if the chosen set is big enough, otherwise the valid formula would be rejected.
- Approaches based on user-provided updates of a transitive closure. This sub-group extends the sub-group of approaches mentioned above in the following way. At a single program point approaches from this sub-group use the approximation of induction in **FOL** to specify the properties of heap structures. On the other hand the updates of the transitive closures are not inferred by a theorem prover but explicitly provided by a user. In this way a relation between the heap state before and after the statement execution is always defined precisely, which is crucial for the verification of a statement's effects on the heap.
- Approaches based on user-provided inductive proofs. Approaches from this group require a user to explicitly provide all necessary inductive proofs and explicitly specify all program points where these proofs are necessary. The validity of user-provided inductive proofs is checked by a **FOL** theorem prover.

• **Inductive theorem provers.** In this group we consider tools which was specifically designed to deal with inductive proofs.

• **Approaches based on an interactive theorem prover.** Approaches from this group use various higher order proof assistants. The approaches work in the following way. A user provides the complete proof, including inductive proofs, that a given program satisfies the given specification, and the proof assistant checks the correctness of the provided proof.

• **Mixed approaches.** Approaches from this group are based on a mix of some of the above approaches.

7.1 Approaches based on decidable procedures

The main advantage of decidable procedures is termination. For any given verification problem the procedure will eventually terminate. It is still possible that verification of some programs is not feasible because of a high complexity of the decidable procedure. Nevertheless it is much better than semidecidable procedures which do not guarantee termination at all.

Another advantage of decidable procedures is a low specification overhead. Some of the approaches which we consider in this section can infer most of the specifications including loop invariants and methods pre- and post- conditions. The last property of the decidable procedures is extremely valuable for a real life verification application.

The advantages of decidable procedures which are mentioned above come at a price. The properties which can be verified by decidable procedures are limited and do not cover the essential part of desired heap properties. To be precise most of the existing decidable procedures are targeted towards the verification of limited properties of various linked lists and trees. Most of the tools are oriented on the verification of shape preservation properties (e.g., the reversal of a list is a list) but cannot say much about how exactly a heap changes (e.g., how the elements of the reversed list are related to the original list).

There is strong evidence that it would be extremely hard or even impossible to overcome these limitations. In the next subsection we provide references on publications which support this claim. On the other hand, the stereotype based methodology is able to precisely describe the updates of arbitrary complexity of any heap structure. This flexibility comes at the cost of specification overhead. The user is responsible for providing this extra specification.

7.1.1 Limits of decidable procedures

In this subsection we consider the limits of decidable procedures.

A natural representation of a heap is a finite graph where a node represents an allocated object and an edge corresponds to a field of a reference type. Therefore a heap property can be represented as a set of finite graphs. In [141] it is proven that the first order theory of the class of finite graphs is undecidable. [35] extends the undecidability result to the class of finite graphs of degree at most 3 and the class of finite planar graphs of degree at most 3.

[119, 92] investigate decidability of may-alias and must-alias in languages with loops and dynamic storage. It is proven that the may-alias relation is not recursive (semidecidable), while the must-alias relation is not even recursively enumerable (co-semidecidable).

Formulas of **FOL** can be classified by the number of variables used in the formula. A fragment of **FOL** which allows only k variables is denoted as FO^k. $\mathbf{FOL} = \bigcup_{k=1}^{\infty} FO^k$. [57] proves that FO^2 is decidable but FO^3 is not. To illustrate the expressive power of FO^2 and FO^3 let us consider the following property. There is a path of length n between two given nodes of a graph, where n is a constant. This property can be expressed for arbitrary n in FO^3 but not in FO^2. What is even more interesting for us is that according to [58] FO^2 with transitive closure (denoted as $FO^2(TC)$), and FO^2 with deterministic transitive closure (denoted as $FO^2(DTC)$) are undecidable for both finite and infinite models. Deterministic transitive closure is the restriction of transitive closure to paths that have no choices. The determinization of the binary relation $r(x, y)$ is defined as $r(x, y) \vee \forall\, z : (r(x, z) \Rightarrow z = y)$. The deterministic transitive closure of the binary relation r is the transitive closure of the determinization of r.

[67] investigates the boundary between the decidability and the undecidability for transitive-closure logics. The paper introduces a rather weak language called $\exists\forall(DTC^+[E])$ that goes beyond trees. It includes a version of transitive closure, and is decidable. $\exists\forall(DTC^+[E])$ is defined as a subset of **FOL** with deterministic transitive closure which satisfies the following constraints:

- In prenex form of each formula all existential quantifiers have to precede all universal quantifiers.
- The language consists of constants, unary relation symbols, and only one binary relation E.
- All applications of DTC are positive occurrence of the form $DTC[E]$.
- Occurrence of $TC[\varphi]$ is allowed only for quantifier-free φ and only in negative positions.

In [67] is proven that $\exists\forall(DTC^+[E])$ is decidable for both finite and infinite models. The authors of the paper show furthermore that essentially any reasonable extension of $\exists\forall(DTC^+[E])$ is undecidable. Any of the following changes causes undecidability of $\exists\forall(DTC^+[E])$: the use of TC; the presence of more than one binary relation symbol; or a single positive use of $DTC[\sigma]$, where σ is quantifier-free.

[88] proves the undecidability of regular graph constraint entailment. A regular graph constraint is a graph representing the heap summary; a heap satisfies a constraint if and only if the heap can be homomorphically mapped to the summary. Regular graph constraints form a very simple and natural fragment of existential monadic second-order logic over graphs.

The central idea of the regular graph constraint is the representation of an unbounded set of heap objects by a single graph node. In shape analysis [127] such nodes are called summary nodes, in role analysis [87] they are called off stage nodes, in separation logic abstract interpretation [56] they are represented by inductive predicates.

The undecidability result implies that there is no complete algorithm which does static checking of procedure pre-conditions and post-conditions, simplify static analysis results, or check that given analysis results are correct.

[149] demonstrates how theorem provers can be used to improve the precision of an abstract interpretation of the heap. The idea behind the

approach is the following: the states of the abstract interpretation represent a set of finite graphs of an unbounded size. Transformations are described by logical formulas. A theorem prover is used to find the most precise abstract value which approximates a result of a transformation application for a given abstract value. If the logic is decidable then the procedure guarantees the best approximation, otherwise the calls to the theorem prover are terminated after a timeout and the procedure returns a sound but not necessarily the most precise result of the transformation.

The approach demonstrates the limits of the heap abstract interpretation. The precision of a single step transformation of a heap abstract interpretation cannot outdo the precision of decidable procedures. Since decidable theories are mostly limited to various linked lists and trees, the same is true for the precision of abstract interpretation.

7.1.2 Type systems for alias control

7.1.2.1 Ownership

To demonstrate the ideas behind type systems let us consider ownership type systems [42, 4, 30]. The ownership discipline provides a way to structure the heap statically. The cornerstone of the ownership discipline is the ownership relation. The ownership relation is a tree relation between objects. The relation is described either by means of type modifiers or by means of annotations. In the first case the type system is used to guarantee the preservation of the ownership relation. In case of annotations a theorem prover is used to verify the same properties. There are several applications of ownership for heap properties verification.

First of all the ownership discipline can be used to statically define the effects of methods. For instance, it could be checked that a method modifies only the objects which are transitively owned by an owner of the callee. This property can be used to guarantee the preservation of class invariants [77, 102]. Several verification tools were constructed on top of the ownership methodology. For instance, Spec# [7] is a C# verification tool.

Another application of ownership is the verification of the tree shape preservation by a data structure implementation.

The following extensions of ownership were proposed with the intention to improve flexibility:

- combination of the ownership and generic types [117, 41].
- dynamic modification of the ownership hierarchy; ownership transfer [29, 104].
- multiple owners of an object; multiple ownership [26].

The main advantage of the ownership methodology is the simplicity of both specification writing and automatic verification. On the other hand there is significant amount of practical examples which do not fit the ownership methodology. There are several reason for this. First of all, not all programs can be organized in to a tree structure. There are situations when an ownership between the objects can be shared between several objects, or there are even cyclic dependencies and modifications between independent program components. The problem was partially addressed by multiple ownership. Another limitation enforced by the ownership methodology is that an object is modified only directly by its owner or by an object with the same owner. The ownership prevents so-called ”deep updates”, when an object modifies another object which it owns not directly but transitively. This property is violated by some design patterns. For instance, the composite design pattern which we have already discussed is problematic for the ownership methodology.

In contrast to the ownership methodology stereotypes can be used to deal with any kind of the heap topologies (including a cyclic one), and any kind of the updates, for instance ”deep updates” and arbitrary topology modification.

7.1.3 Decidable heap logics

7.1.3.1 Monadic second-order theory of two successors on the model of finite binary sequences

[118] provides a decidable algorithm for the monadic second-order theory of two successors on the model of finite binary sequences. The logic is denoted as $S2S$. The decidability problem of $S2S$ is reduced to the check that a given tree automaton recognizes the empty set. The decidable procedure is constructed for the following model:

- The domain of the model is the set of all finite binary words.
- The functions of the model are two successors. One of them adds 1 and another adds 0 to the end of the given sequence.
- The two predicates of the theory are the prefix and the lexicographical orders.

The logic consists of the above predicates and functions, logical bindings, and quantifications over elements and set of elements. It is also shown that the result can be extended on $n \leq \omega$ successors. The corresponding logic is denoted as SnS. Another extension is quantification over finite sets. The finiteness of the quantified variable is checked via checking the existence of a minimal and a maximal elements of the set.

$S2S$ is very important for software verification. All decidable procedures which we consider in this subsection are less expressive then $S2S$ but have better complexity. The reasons why $S2S$ is so important for software verification is the following one. $S2S$ can express and therefore check the validity of the formulas from the following theories:

- Logic of indexes of a string. The logic consists of: quantification over the indexes and the sets of indexes, order on indexes, sets inclusion. SnS can be used to express the string logic with an alphabet of the size n. When $n = \omega$ the size of the alphabet is unbounded but countable.

- Logic of trees. The logic consists of: quantification over three paths and sets of three paths, path inclusion (prefix order), set inclusion. SnS can be used to express the tree logic of n-branching trees. When $n = \omega$ the branching is unbounded but countable.

[98] proved that the complexity of weak monadic second order logic is not-elementary recursive. In other words, there is no $k \in \mathbb{N}$ such that the complexity of the deciding procedure could be bounded by:

$$2^{2^{\cdot^{\cdot^{2^n}}}} \left.\vphantom{2^{2^{2^n}}}\right\}k$$

It is also proven that the complexities are not elementary for both the satisfiability and the finite-satisfiability, even when restricted to first-order quantification.

[63] describes MONA; a tool for checking validity of monadic second-order logic on string formulas. MONA is based on the translation into finite automata, whose state spaces are represented by binary decision diagrams [1]. The logical models of the logic is finite strings. The language of MONA consists of:

- position terms: the first index, the last index, a position variable, constant increment and decrement.

- position set term: empty set, set of all indexes of the model string, union, intersection, complement, add and remove a constant to all elements of a set.

- formulas, equality and order on indexes, equality and set inclusion on set of indexes, logic bindings, universal and existential quantifiers over index and set of indexes variables.

7.1.3.2 Graph types

[83] introduces graph types. Graph types is an approach to shape invariant specification. A shape invariant is described as a composition of the

graph backbone (which is a canonical spanning tree) and extra edges. The extra edges are specified by the language of regular "routing expressions". A routing expression gives a relative address of the extra edge in the backbone. Both the backbone and the routing expressions of a data structure are described by means of monadic second-order logic.

A program which uses graph types is allowed to modify only the backbone tree but not the extra edges. An update of a backbone edge can violate a shape invariant. Code which rebuilds extra edges to restore the shape invariant is generated from the shape invariant specifications for each backbone edge update (edge addition and removal). Such an update can be constructed automatically only if each routing expression always defines a unique destination. A graph type is called well-formed if and only if each of its routing expressions satisfies the above property. The well-formedness of a graph type is checked statically by the decision procedure for monadic second-order logic.

It is proven that for a well-formed graph type a dynamic update of extra edges takes not more then linear time over the size of the data structure. Specifications for the following data structures are provided: linked list, cyclic-list, doubly-linked list, black and red tree. The following quote from [83] highlights the limitations of graph types: 'Although much can be expressed in the monadic second-order logic on graph types, there are simple operations that cannot. For example, one cannot represent the result of replacing a subtree with another subtree (although certain properties of the result may be expressible).'

[84] describes a shape verification approach based on graph types. A shape invariant is a graph type. A set of finite graphs is described as a monadic second-order logic formula. A transformation of a set of finite graphs is described by so called transductions. A transduction is described as a finite sequence of edge additions and removals. Transductions also have pre-conditions which check that am operation can be executed on a given set of graphs. For instance the following properties are checked: existence of a removed edge, there are no two edges with the same label and source. Transductions are used to define the predicate transformers. The predicate transformers are used for backward propagation of formulas and can be used to define the weakest pre-condition calculus.

In comparison with the stereotype based approach graph types have the following limitations:

- The shape of a data structure is limited to one whose extra edges are deterministic. For instance, general graph, DAG, and planar graph cannot be specified as a graph type.

- Some practically important properties cannot be expressed in monadic second-order logic. For instance we cannot precisely describe how the shape of a tree will change after an edge removal or addition. As result there is no way to provide the precise enough specification for the loop invariants and the method specification. For instance, let us consider a loop which performs a list reversal. We can specify and verify that if the initial data structure is a list then the resulting data structure is also a list and contains exactly the same elements as the initial data structure. On the other hand we cannot specify and therefore verify that the resulting list is the reverse of the initial one. The relation between indexes of the elements in the initial and in the resulting list remains unspecified.

[72] describes a verification technique for various linked lists which are based on [84]. The specification language is monadic second-order logic. The allowed structures are linked lists extended with extra edges, which are specified by routing expressions. The approach uses the weakest precondition calculus to reduce the problem to checking the proof obligations in monadic second-order logic. Limitations of the technique are the same as already mentioned for [84]. [99] extends [72] to all graph types.

7.1.3.3 Verification via structure simulation

[68] demonstrates an approach to encoding singly-linked lists, binary trees, unbounded trees, and doubly-linked trees in monadic second-order logic. The paper provides an example where a shape of an unbounded number of nodes of a doubly-linked list is temporarily violated and later restored.

7.1.3.4 Logic of reachability expressions

[10] introduces the "logic of reachability expressions". It is a logic for describing linked data structures. Decidability is achieved by translation into monadic second-order logic. Examples are a singly-linked list reversed and deletion of an element from a list.

7.1.3.5 Logic of interpreted sets and bounded quantification

[90] presents the "logic of interpreted sets and bounded quantification". The logic can be used to specify the properties of heap manipulating programs. The logic consists of:

- unary uninterpreted functions
- boolean logical operations
- both existential and universal quantifiers but not quantifier alternation
- ternary predicates $x \xrightarrow{f} y \xrightarrow{f} z$ "reachable via intermediate value". The predicate specifies that x reaches y via transitive closure of the unary function f and y reaches z via transitive closure f.

- ternary functions $Btwn(f, x, y)$ "set of objects between x and y". y belongs to $Btwn(f, x, z)$ if and only if $x \xrightarrow{f} y \xrightarrow{f} z$.
- a function inversion.

There are several syntactic limitations over formulas. For instance the same functional symbol cannot be used in $f^{-1}(x)$ and in $Btwn(f, x, z)$.

A decision procedure for the logic is provided. The inference rules of the decision procedure are encoded in **Boogie** (which we describe below) as axioms with triggers. An **SMT** theorem prover was used to verify the generated proof obligations.

Properties of doubly-liked lists cannot be expressed in the logic due to the syntactic limitations over formulas. Such properties were still encoded and verified by Z3. In the general case termination is not guaranteed, but in some cases extra triggers can guarantee termination.

Monadic second-order logic can express more complex shape properties than what it is allowed by "logic of interpreted sets and bounded quantification".

7.1.4 Abstract interpretation of heap

In this subsection we consider approaches to heap verification which are based on abstract interpretation [36]. Abstract-interpretation represents possibly infinite sets of program states with some finite abstract values. An iterative computation is carried out to determine an appropriate abstract value for each program point. The result of the computation is an abstract value that summarizes the sets of the reachable concrete states at each program point. Since we are interested in the verification of heap properties, the abstract values of the above analysis represent possibly infinite sets of heaps.

7.1.4.1 Shape analysis

[127, 126] introduce shape analysis, an abstract interpretation analysis of heap properties. A single memory state is represented as a finite 2-value logic structure, where elements of the structure represent the allocated objects and relations represent the references between the objects. Additional predicates are used to represent the extra properties, e.g., readability from other nodes, the number of input references is less or equal to one, and so on. Finite 3-valued logical structures are used to represent a potentially infinite set of memory structures. A special unary predicate is used to designate nodes that may represent more than one individual from a given 2-valued structure. Such nodes are called summary. The relation between 2-valued and 3-valued structures is defined in the following way. A 3-value structure S represents a 2-value structure $S^{\#}$ if and only if:

- There is a surjective function f from elements of $S^{\#}$ into elements of S.
- For every predicate p and elements $u_1, \ldots, u_n$ of $S^{\#}$ either the interpretation of $p(f(u_1), \ldots, f(u_n))$ on S is equal to 1/2 (unknown predicate value) or it is equal to the interpretation of $p(u_1, \ldots, u_n)$ of $S^{\#}$.

Properties of structures are represented by formulas in first-order logic with transitive closure and equality, but without functional symbols and constants. The "embellishing" theorem states that any formula that evaluates to a definite value in a 3-valued structure evaluates to the same value in all of the 2-valued structures embedded (represent by) into that structure. Transformations are represented by formulas which relate the values of predicates in the initial and transformed structures. A fix-point computation is used to infer loop invariants and missing method specifications. To improve the precision of the abstract interpretation the specification writer has to introduce additional predicates which capture extra properties.

The approach is implemented in a tool called TVLA.

[95] describes the application of TVLA to the verification list properties. The tool is precise enough to verify that bubble-sort and insertion-sort produce sorted lists. Also it was verified that element-insertion, element-deletion, and merging of two sorted lists preserve the "is-sorted" invariant. The tool was also able to identify an error in the erroneous versions of the bubble- and insertion-sort procedures.

[97] demonstrates how shape analysis can be enriched with ideas from separation logic. The authors present a new abstraction based on decomposing graphs into sets of subgraphs. The new abstraction leads to a small loss of precision, while yielding substantial improvements to efficiency.

[122] demonstrates how shape analysis can be enriched with ideas from ownership. A program is split into modules. Shape analysis is used to verify that references between modules form a tree.

7.1.4.2 Separation logic

[114] describes separation logic; an approach for heap specification. The approach is based on the following two ideas:

- "separating conjunction" $P * Q$, which asserts that P and Q hold for separate parts of a data structure. The conjunction provides a way to compose assertions that refer to the different areas of memory, while retaining disjointness information for each of the conjuncts. The semantics of $P * Q$ is defined via existential heap quantification in the following way. $h \models P * Q$ if and only if $\exists h_0, h_1 : h_0 \# h_1, h_0 * h_1 = h, s, h_0 \models P \wedge h_1 \models Q$. Where:
  - $h \models \varphi$ means that an assertion φ holds on a heap h

  - $h\#h'$ denotes that the domains of heaps h and h' are disjoint.
  - $h * h'$ denotes the union of disjoint heaps.

- inductive definitions of predicates are used to define data structures. Definitions for trees and the linked lists are provided.

An advantage of the approach is that a description in separation logic provides a natural way for a heap decomposition. A disjunction can be used to describe a heap from the different perspectives. In the stereotype based approach this corresponds to a description of a heap state as a combination of several stereotype slices.

[56] describes an interprocedural shape analysis based on separation logic. The approach is implemented in the SUMMATE tool. The tool implements a fixed-point computation over an abstract domain built from assertions expressed in separation logic. The analysis represents sets of heaps as a separation logic formula. The formula describes a heap as a disjoint union of linked lists of unknown size and single allocated locations. To achieve termination of the fixed-point computation, the number of elements in the disjoint union is limited. The approach is similar to shape analysis but works on a more specific domain, and therefore has potential to achieve better performance.

[14] introduces the tool SLAyer. The described approach extends [56] by covering binary trees. [12] extends the approach to lists of lists. The approach was used to verify the IEEE 1394 (firewire) Windows driver. [148] proposes a more precise list approximation by distinguishing possibly empty and non-empty lists. The paper also proposes a new join operation for the separation domain which aggressively abstracts information for scalability yet does not lead to false errors. The shape properties of Windows and Linux device drivers (firewire, pci-driver, cdrom, md, etc.) were verified.

The following tools are based on abstract interpretation of separation logic:

- [44] ”Space Invader” for C.
- [25] ”Infer” for C.
- [13] ”Smallfoot” for C.
- [14] ”SLAyer” for C.
- [45] ”jStar” for Java.
- [143] ”MultiStar” for Eiffel; based on ”jStar”.

As it explained in the **Subsection 7.1.1** the precision of a single step transformation of a heap abstract interpretation cannot outdo the precision of decidable procedures. Since decidable theories are mostly limited to various linked lists and trees, the same is true for the precision of abstract interpretation. On the other hand stereotypes can be used to verify arbitrary properties of any heap structure.

7.1.4.3 Role analysis

[87] introduces another abstract interpretation approach to the verification of heap properties, "role analysis". The central concept of the approach is the role concept. Each object has a single role at any program point. A role can change over time. Each role definition specifies the constraints that an object must satisfy to play the role. Field constraints specify the role of the objects to which the fields refer, while slot constraints identify the number and the kind of aliases of an object. Procedures specify the initial and final roles of their parameters. Each procedure also specifies its read and write effects.

At every program point the set of all heap objects can be partitioned into: onstage objects referenced by a local variable or parameter; offstage unreferenced by a local or parameter. Onstage objects do not need to have correct roles. Offstage objects must have correct roles assuming some role assignment for onstage objects.

The analysis representation is a graph in which nodes represent the objects and edges represent the references between objects. There are two kinds of nodes: onstage nodes representing one onstage object; and offstage node, with each offstage node representing a set of objects that play that role. The role consistency for a heap can be verified incrementally by ensuring role consistency for every node when it goes offstage.

Glue invariant states in our methodology were inspired by roles.

The role analysis is an abstract interpretation and has the same limitations as any other abstract interpretation. It is mostly limited to various linked lists and trees (see **Subsection 7.1.1**). On the other hand stereotypes can be used to verify arbitrary properties of any heap structure.

7.2 Approaches based on automatic theorem proving

In this section we consider approaches based on automatic theorem provers. As we have mentioned above the main problem which has to be addressed by such approaches is the approximation of transitive closure and inductive reasoning. These properties cannot be expressed in **FOL**. In general we need second order logic (**SOL**) to express them. Nevertheless many of them can be expressed in monadic second-order logic (**MSOL**). We can think about **MSOL** either as about a subset of **SOL** where the second order quantification is allowed only via unary relations, or as an extension of **FOL** with quantification over sets.

[34] investigates the expressivity of **MSOL** for the description of (possibly unbounded) sets of graphs and their transformations. For instance, it was proven that the following properties of vertices x, y, sets of vertices X, and an arbitrary directed graph G can be expressed in **MSOL**:

- $x = y$ or there is a nonempty path from x to y.
- G is strongly connected.
- G is connected.
- $x \neq y$ and there is a path from x to y, all vertices of which belong to set X.
- X is a connected component of G
- G has no circuit
- G is a directed tree
- G has no circuit, $x \neq y$ and X is the set of vertices of a path from x to y
- G is planar.

Nevertheless there are some properties which cannot be expressed in **MSOL**. For instance, the following properties can't be expressed in **MSOL**

- two sets X and Y have equal cardinality
- a directed graph is Hamiltonian
- in a directed graph a set X of vertices is the set of vertices of a path from x to y
- a graph has a nontrivial automorphism.

[34] also investigates which transformations of the sets of graphs can be expressed as **MSOL** formulas. Such **MSOL** formulas define the values of relations after the transformation in terms of relations before the transformation. It is shown that for each such transformation and formula β it is possible to construct a formula β' such that β holds after the transformation if and only if β' holds before the transformation. This observation can be used to construct the weakest pre-condition calculus. Nevertheless there are some transformations of graph sets which cannot be expressed in **MSOL**. The following limitations of **MSOL**-definable transductions are proven:

- The image of an **MSOL**-definable class of structures under a definable transformation is not **MSOL**-definable in general.

- The inverse of a definable transformation is a transformation that is not definable in general.

- The intersection of two definable transformations is a transduction that is not definable in general.

As we can see even **MSOL** cannot express all interesting heap properties. On the other hand most automatic theorem provers can deal only with **FOL** which is even less expressive then **MSOL**. The last observation illustrates why dealing with heap specifications is challenging for state-of-the-art automatic theorem provers.

Before we move to the consideration of approaches to heap verification which are based on automatic theorem provers let us briefly consider the current state of art of automatic theorem proving.

7.2.1 Automatic theorem provers for FOL

Currently there are two promising types of automatic **FOL** theorem provers.

The first one is tableaux [37] or resolution based [27] first order logic (**FOL**) theorem provers. For instance the following theorem provers use this approach: Vampire [121], LEO-II [11] (it covers even a subset of higher-order logic) , iProver [85], Spass [144], and many others. The second type of automatic theorem provers is satisfiability modulo-theories (**SMT**) theorem provers, which are usually based on the Davis-Putnam-Logemann-Loveland (**DPLL**) [112] algorithm or on the Nelson-Oppen method [140, 111]. For instance the following theorem provers use this approach: Z3 [39], Simplify [40], CVC3 [8], MathSAT [21], and many others.

Tableaux and resolution based theorem provers were initially designed to cover the entire **FOL**. Most of these theorem provers guarantee completeness; each sound **FOL** theorem will eventually be proved by the theorem prover. For unsound theorems termination is not guaranteed. Completeness is an extremely nice property from a theoretical perspective. Nevertheless from a practical perspective performance is more important than completeness. Therefore some tools sacrifice completeness in order to achieve better performance on the practically valuable subset of **FOL** formulas.

SMT theorem provers were initially designed as a combination of several decidable procedures with a **SAT** solver. A set of decidable procedures usually includes: real and integer arithmetic, bit-vectors, arrays, records, uninterpreted functions. The resulting procedure is decidable but it is only able to deal with quantifier-free formulas. To overcome this limitation **SMT** provers were extended with heuristics for quantifier handling [101, 51], which in general do not guarantee completeness. Overall for a given formula with quantifiers they try to identify a variable substitution such that the resulting quantifier-free formula can be validated by the underling procedure.

The ability to deal efficiently with the basic programming theories, which we have mentioned above, is essential for software verification. Therefore most of the approaches which we consider in this section use **SMT** theorem provers.

7.2.1.1 Boogie

Some approaches which we consider in this section use automatic theorem provers to verify proof obligations generated by the weakest pre-condition calculus. Since the weakest pre-condition calculus is pretty standard it does not make sense to re-implement it for any tool. Instead the intermediate verification language **Boogie** [75] can be used. In this case heap verification technique can be implemented as a translator of a given verification problem into **Boogie**. **Boogie** uses the weakest pre-condition calculus to generate proof obligations which are then verified by an automatic theorem prover. In most cases Z3 [39] is used for this purpose.

Boogie describes a control flow graph by means of sequential composition, nondeterministic branching, and goto statements. Later on while loops and if statements were added to the language. The main specification primitives are ”assert” and ”assume” clauses. ”assert” clause checks that a given formula holds at the current point of the control flow graph, while ”assume” clause adds a given formula to premisses. **Boogie** also provides pre- and post- conditions which can be expressed by means of assume and assert clauses, respectively. The types provided by **Boogie** include integer, boolean, and map types. Boogie does not provide a native heap specification, but a heap can be specified as a polymorphic map from a location to a value.

7.2.2 Approximation of induction in FOL

7.2.2.1 Simulating reachability using first-order logic

[94] provides an approach to the approximation of the transitive closure relation. The authors describe three induction schemes which capture the properties of transitive closure of a given binary relation. Each scheme is parameterized by one or two variables of the set type. The authors provide a heuristic which for a given formula φ from **FOL** with transitive closure instantiates the inductive schemes with **FOL** descriptions of set variables. Let us denote the instantiation of inductive schemes as ϕ. ϕ is a sound but not complete description of a transitive closure in **FOL**. In other words ϕ is an approximation of a transitive closure in **FOL**. The soundness of the approximation for any instantiation is guaranteed by the meta-proof. A **FOL** theorem prover can be used to verify that $\phi \Rightarrow \varphi$. The authors used the approach to verify an implementation of the in-place reversal of a singly-linked list in the SPASS theorem prover [144].

The main advantage of the approach is the expressivity and the absence of specification overhead. The approach attempts to verify formulas from **FOL** with transitive closure without any specification overhead. The main disadvantage of the approach is low predictability. It is hard to predict what properties can be verified by the approach. For instance, one of the provided induction schemes is specifically targeted towards dealing with updates of transitive closure but far from being complete. On the other hand, the stereotypes based approach uses user-provided specifications to precisely describe the transformation of transitive closures during a program execution.

7.2.3 User provided updates of a transitive closure

In this subsection we consider approaches to heap verification which are based on user-provided specifications of updates of a transitive closure. Stereotypes-based verification is one of these techniques.

The main idea behind these approaches is the following one. Let us assume that a class of a verified program contains a field of reference type f. Let us also assume that to specify the program we need the transitive closure of f. To represent the transitive closure of f we introduce a ghost field f_{TC} of type set of references. The value of the ghost filed f_{TC} of object o has to contain the objects reachable via the transitive closure of f starting from o. Since the transitive closure is not expressible in **FOL**, f_{TC} cannot be specified precisely. Therefore the values of f_{TC} are approximated by **FOL** formulas. This part of the approach is the same as the one which is described in the previous subsection. What is done differently is the reevaluation of f_{TC} during the program execution.

Values of f change during the program execution. Therefore f_{TC} has to be updated accordingly. The old and the new values of f_{TC} correspond to the transitive closure of two distinct relations. Therefore an induction has to be used to establish the relation between the old and the new values of f_{TC}. On the other hand induction is not a part of **FOL** and therefore can hardly by handled by automatic theorem provers. Therefore instead of inferring the relation between the old and the new values of f_{TC} the user is asked to provide it explicitly. For every update of a value of the field f the user has to provide ghost updates of f_{TC} of all affected objects.

Another question which arises is what kind of transformations and for which heap structures can be specified this way in **FOL**. The question is partially addressed by the following two papers [47, 46]. The papers address the following problem. Suppose G is a graph and TC_G is its transitive closure. If G' is a new graph obtained from G by inserting or deleting an edge e, can the new transitive closure $TC_{G'}$ be defined in first-order logic using G, TC_G and e? It is shown that the answer is positive for

1. acyclic graphs (DAGs),

2. graphs where the vertices of the deleted edges are not in the same strongly connected component,
3. graphs in which not more then one path between each pair of vertices exists (0-1-path graphs).

It was also proven that for the insertion case question (transitive closure definability in first-order logic) the answer is yes. It is left open whether the new transitive closure is definable in first-order logic for all graphs which are obtained by removing edges. It is also unclear up to which extent the provided result can be generalized in case of addition and removal of unbounded sets of edges.

7.2.3.1 Dynamic frames

[80, 81] introduce the dynamic frame theory. A dynamic frame is a specification variable whose values are sets of references. The values of the specification variable are defined by an expression that depends on a heap. Dynamic frames can be used to define a frame rule for a given expression. The frame rule states that if the values of locations which belong to a dynamic frame are preserved then the value of the framed expression is also preserved. Dynamic frames and the frame rule are used to specify module frames. The following approach for frame disjointness preservation is presented. If f and g are dynamic frames and g is self framed (g is a frame of g) then any transformation which affects only values of locations from f and adds to f only freshly allocated locations preserves disjointness of f and g.

To specify frame properties of dynamic heap structures dynamic frames have to use transitive closure. Therefore the approach cannot be directly used for automatic verification.

[76] describes Dafny; a verification language which is built on top of Boogie [75] and uses dynamic frames for heap verification. Dafny is targeted on the verification of object-oriented pointer manipulating programs. It natively supports a heap, classes, and methods. Additionally to pre- and post-conditions method specifications include modifies clauses, which specify the set of locations which are possibly affected by the method execution. The types are extended with sets, sequences, and algebraic types. For specification purposes, ghost variables and functions are added. Ghost variables are used in the verification of the program but not needed at the run time. Functions are adjusted with measures which guarantee their termination. The axioms which specify functions are generated from the functions bodies.

7.2.3.2 Region logic

[5] introduces region logic. A main specification primitive of region logic is a region. A region is a ghost field of the type set of non-null references. The regions specification language is a simple set theory which consists of:

- region inclusion
- region disjointness
- inclusion of a region image by another region
- disjointness of a region image and another region
- equality of a value of a region type to another region
- quantification over references and integers
- logical bindings

Regions can be used to express behavioral and topological specification. One of the central applications is transitive closure approximation, e.g., a set of objects reachable via a linked list. A specification provider is responsible for instrumenting the source code with region updates. The main advantage of the approach is that the desired specifications can be expressed in **FOL** and verified by **SMT** theorem provers. The paper demonstrates the region logic approach to specification and verification on the example of the observer design pattern.

[3] describes how the region logic approach can be expressed in **Boogie** and verified by Z3. A partial specification of list copy is provided. The examples specify that the input list is disjoint from the output list. The second provided example is the observer design pattern.

[123] describes verification of the composite pattern with region logic. The paper also introduces the "VERL" tool which is built on top of **Boogie**.

[109] describes an invariant verification methodology based on region logic. All classes are split into disjoint modules. Invariants are associated with a module but not with a class. Each module is adjusted with dynamic boundaries. Dynamic boundaries are read effects which guarantee the module invariant preservation. A client of a module is obliged to respect the module's dynamic boundaries. Therefore the client preserves invariants of the module even without knowing them. Regions are used to specify a module's dynamic boundaries and invariants.

Our approach is inspired by region logic. Our approach extends region logics in the following way:

- It enabling the reuse of specifications, ghost updates, and proofs.
- It provides a systematic approach to ghost updates and method specifications. A universal transformation for a given stereotype can be used to express an arbitrary ghost update and method specification.
- It improves readability of ghost updates and specifications.

7.2.3.3 Verifying properties of well-founded linked lists

[91] describes an approach to the specification of cyclic and acyclic linked lists. The approach is very similar to our stereotype based verification. Similarly to our approach the authors use ghost fields (regions) to approximate the transitive closure and user provided ghost updates to describe how regions evolve over time. In terms of stereotype based verification the provided methodology covers the following parts of the sequence stereotype:

- stereotype items: right tail and head of the list.
- stereotype invariants: 5 properties; a subset of sequence stereotype invariants.
- stereotype operations: only basic operations (addition and removal of a single reference) are provided. The specification of loops and recursive procedures which perform lists modifications require explicit updates of stereotype items.
- glue invariant. A user has to explicitly specify which fields form lists and which objects belong to the lists. Under this assumption ghost updates of regions are inserted automatically.

An on-paper meta-proof guarantees that regions indeed approximate the desired transitive closure.

In contrast to the proposed approach stereotypes can be used to verify arbitrary property of any heap structure. We can think about stereotype as an extension of the approach proposed in [91] to deal not only with linked lists but with arbitrary heap structures as well.

7.2.4 User provided inductive proofs

In this section we consider tools which rely on user-provided inductive proofs to verify heap properties with automatic theorem provers. The obvious disadvantage of this approach is extra work which has to be done to perform verification. Another limitation of the approach is that the user has to be qualified enough to be able to produce the inductive proofs, and therefore the tool cannot be used by an average software developer.

Stereotypes based methodology can be used to avoid inductive proofs. It could be achieved in the following way. Instead of inductive definitions, corresponding stereotypes have to be used to specify heap structures. The user has to annotate the program with stereotype operations. The desired properties of heap structures are inferred from stereotype invariants and stereotype operation calls.

7.2.4.1 VeriFast

[70] describes VeriFast, a verification tool for C. The proposed methodology is a combination of user-provided inductive proofs and separation logic. A heap is represented in the separation logic style as a disjoint union of single memory locations and inductively defined data types. To verify a program the client has to explicitly prove the desired properties of the inductive data structures. Quite often these proofs require induction. A user has to explicitly annotate the source code with lemmas which have to be used during the verification as well. For instance, to verify that a method "add" of a linked list preserves the list shape, a client has to provide an inductive proof that an addition of the element preserves the list shape and then annotate the source code of the method "add" with the proved lemma. The method is flexible but requires a lot of extra work. [71] describes verification of the composite pattern which is represented as a binary tree with VeriFast.

In contrast to VeriFast stereotypes provide a systematic approach to a specifications and proofs reuse. Most of the proofs have to be done during the stereotype construction. Later on these proofs can be reused to verify real heap structures.

7.2.4.2 Implicit dynamic frames

[138] proposes an approach to the verification of object-oriented programs. According to the approach each method specification has to include a method footprint. A method footprint is an upper bound of the memory locations which are read or written by the corresponding method execution. Footprints are specified by means of pure methods that return sets of memory locations. Such pure methods are called dynamic frames.

[133, 135] extend [138] by adding a footprint inference mechanism. The footprint of a method is inferred from the method pre-condition. A callee can only read or modify an existing location if the location is mentioned in its pre-condition.

[134] describes the implementation of the implicit dynamic frames approach in the Java verification tool VeriCool.

[136, 137] propose a variant of [134] where a symbolic execution instead of the weakest pre-condition calculus is used. The heap representation is similar to the one used in [70].

Implicit dynamic frames is an extension of dynamic frames and has similar limitations. They do not provide a systematic approach to the verification of dynamic heap structures. The latest third version of the VeriVool tool uses two approaches to address this issue:

- The approach of region logic, by introducing ghost fields of type set of referencee and ghost updates.

- The approach of VeriFast [70] via induction of inductive definitions and user-provided inductive proofs.

[78] describes Chalice; another tool which is built on top of **Boogie**. The tool is targeted on the verification of concurrent object-oriented programs. Here we consider only the specification infrastructure provided by Chalice for the specification of heap structures, and ignore the concurrency related issues. The specification and verification of heap structures is inspired by implicit dynamic frames. Data shapes are described by means of recursive predicates and single location access. For instance, a list predicate can be specified as access to the list head in conjunction with the list predicate for the tail. To avoid the explicit usage of recursive predicate definitions a client is provided with the statements **fold** and **unfold**. **fold** checks that the predicate definition holds at the current program point, consumes the permissions required by the definition, and produces the predicate. The statement **unfold** consumes the predicate and returns its definition.

To prove properties of heap structures, intensive usage of **fold** and **unfold** is required. In some sense these extra annotations are similar to inductive proofs. For instance, to add a node to the end of the list it is necessary to unfold the list, add the new element, and fold it again.

7.3 Inductive theorem provers

According to Gödel's first incompleteness theorem [53] for any formal theory of arithmetic there will be formulas which are true but unprovable. The results of Gödel introduce infinite branching points into the search space and show that it is impossible to build a complete inductive theorem prover. The main problem that automatic inductive provers have to address is infinite branching.

The source of infinite branching is the identification of a well-founded order which can be used by the inductive proof. For any non-trivial recursive data type there will be an infinite variety of different well-founded orders. No computer program can generate them all.

Another source of infinite branching is the cut elimination rule. It can be formulated in the following way:

$$\frac{A, ? \vdash \Delta, \ \ ? \vdash A}{? \vdash \Delta}$$

A is called a cut formula. Since any formula can be used as a cut formula, an application of the cut elimination rule results in infinite branching. The proof of the cut elimination theorem can be found in [52]. The cut elimination theorem states that the cut rule is redundant for first order theories. Unfortunately cut elimination does not hold for inductive theories (see [86]). The cut rule cannot be eliminated and it is a source of infinite branching.

There are two major approaches to the automation of inductive proofs:

- Explicit induction
- Proof by consistency

7.3.1 Explicit induction

Probably the most well known automatic explicit inductive theorem prover is Nqthm [19]. It is also known as the Boyer/Moore theorem prover. It is one of the first theorem provers which was designed to handle inductive proofs automatically. Later on Nqthm was re-implemented as ACL2 [82].

It has been successfully applied to prove complicated mathematical theorems and hardware verification (e.g. verification of the kernel of the AMD5K86 floating-point division algorithm [100]).

Both Nqthm and ACL2 use a simple, sub-first-order, type-less logic, based on primitive recursive arithmetic adapted from numbers to lists. Variables are treated as implicitly universally quantified; therefore there is no existential quantification.

Other theorem provers which use the explicit induction approach to prove inductive theorems are INKA [66] and Oyster/CLAM [23].

Let us consider the explicit induction theorem proving approach on the example of ACL2. The ACL2 approach to the automatization of induction proofs is based on the similarity of recursion and induction. The ACL induction proof analysis consists of two steps:

1. Analyze each recursive function in isolation:
   (a) Identifies the measure and the order according to which the recursive function is well-defined (always terminates). ACL uses lexicographic order over vectors of natural numbers as the well-founded orders.
   (b) Identify the inductions which the function suggests.
2. Combine the inductions suggested by the recursive functions mentioned in a formula which is proved by induction. The combined induction has to be suitable to prove the whole formula by induction.

A recursive function in ACL can be represented as a set of recursive calls guarded by conditional statements. In ACL definitions of recursive functions in this form are called *machines*. The structure of a recursive function is used to identify both the measure and the inductions suggested by the function definition.

The measure identification relies on user-provided induction lemmas. The general form of the induction lemma is the following
$\varphi \Rightarrow m(t_1, \ldots, t_n) < m(t'_1, \ldots, t'_n)$. Here m is a measure and φ is a guarded

condition which guarantees the validity of the lemma. An example of an induction lemma is $x \neq 0 \Rightarrow x - 1 < x$.

Identification of a measure of a recursive function is organized in the following way. If

- $f(x_1, \ldots, x_m)$ is a recursive function,
- ϕ is a guard formula of a recursive call $f(exp_1, \ldots, exp_m)$,
- $\varphi \Rightarrow m(t_1, \ldots, t_n) < m(t'_1, \ldots, t'_n)$ is an induction lemma,
- $\phi \Rightarrow \varphi$ holds,
- $i_1, \ldots, i_n \subseteq [1..n]$ is a set of variables which are used to guarantee termination of the function,
- $t_1, \ldots, t_n, t'_1, \ldots, t'_n$ matches as syntactical terms with $exp_1, \ldots, exp_m$, $x_1, \ldots, x_m$

then m is suggested as a measure for the recursive call. Lexicographical order can be used to combine several measures into one. The measure for the whole function is constructed by a combination of measures of recursive calls.

A recursive function definition together with the measure suggest an induction scheme. The case split of the induction is based on the machine of the function (the function definition reduced to the special form). The induction order is based on the function measure. If recursive functions $f_1, \ldots, f_n$ are mentioned in a formula which we try to prove by induction then the inductive scheme for the whole formula is constructed by applying the following heuristics to the inductive schemes by $f_1, \ldots, f_n$:

1. Subsumption of induction schemes. An induction scheme s_1 is subsumed by an induction scheme s_2 if the cases of s_1 are a subset of the cases of s_2.

2. Merging of induction schemes. When we merge s_1 and s_2 some of the cases of s_1 are subsumed by s_2 and vice versa.

3. Flawed induction schemes. s_1 is flawed if there is another scheme s_2, such that some variables of s_2 are induction variables of s_1. If any scheme is unflawed, we throw all flawed schemes out. If all schemes are flawed, we simply do not throw out any and proceed.

4. Tie breaking rules. Choose the most nested induction scheme. We choose the scheme that is credited with the largest number of terms that are not primitive-recursive. If we still have a choice we choose arbitrarily.

5. Superimposing the machine. Merge equivalent cases.

The result of the application of the generated induction scheme to the formula has to be proven by a first order logic theorem prover.

Limitations of the explicit induction can be described by the following quote from [22]. ”The difficulty of the search control problems that arise in inductive theorem proving causes all current automatic provers to fail on some apparently simple conjectures. Until the technology is significantly improved it is, therefore, necessary to involve a human user in assisting with proof search.” In other words there are still plenty of situations when the user has to identify the induction scheme.

In comparison with the stereotype based approach we can say that the explicit induction is more generic but less efficient. The explicit induction can be used to prove induction hypothesis in any theories while the stereotypes are specifically targeted to deal with heap structures. Nevertheless due to this specialization, stereotypes can exploit knowledge about the domain (heap structures) and be more efficient. For instance, we distinguish theorems which have to be proven to describe the heap at a single program point and theorems which describe transitions between program points. For the last type of theorems we introduce a special machinery: **SLS** and universal transformations. In this way we avoid user-provided inductive proofs in regards to heap transformations.

Another clear advantage of stereotypes is reusability. As soon as we have one or more stereotypes we can combine them to describe new heap structures. We can use theorems which are proven about stereotypes (e.g., the stereotype invariant preservation by the stereotype universal transformation) to prove new theorems about new heap structures.

7.3.2 Proof by consistency

One of the best known automatic inductive theorem provers which uses proof by consistency is RRL (Rewrite Rule Laboratory) [79]. RRL is an automated reasoning program based on rewriting techniques. RRL includes implementations of various rewriting-based approaches to proving first order logic formulas and methods for proving first-order equational formulas by induction.

Proof by consistency is also known as *inductionless induction.* The main idea behind the approach (as it described at [32]) is the following.

Let us assume that we want to prove an inductive theorem T in a theory E. The first limitation which is introduced by the approach is that it considers not all but only Herbrand models of the theory E. A Herbrand model of the theory E is a model whose domain is the Herbrand universe. The Herbrand universe is the set of all ground terms, constructed from functional symbols of the theory E. In other words each element of the Herbrand model of the theory E can be represented as a ground term. If we are interested in inductive theorems of E which hold in non-Herbrand models then we cannot use inductionless induction to prove them. Let us consider an example that explains this constraint.

Let us consider the theory E which describes some of the properties of natural numbers. The set of functional symbols of E is $\{0, s, +\}$. The axioms of E are $0+x=x$ and $s(x)+y=s(x+y)$. Let us now consider the formula $x+0=x$. $x+0=x$ is not an inductive theorem of E. Let us construct the structure which is a model for E but not for $x+0=x$. The domain of the model is $\{0, a\}$. The interpretation of 0 is 0. The interpretation of s is the identity ($s(x)=x$), and the interpretation of the addition can be described in the following way $a+b=b$. Let us check that the described structure is a model of E. The first axiom follows immediately from the definition of the addition. The validity of the second axiom can be checked in the following way $s(x)+y=y=s(y)=s(x+y)$. $a+0=0$ is not equal to a. Therefore the structure is not a model for $0+x=x$. On the other hand if we consider only Herbrand models (where each element can be represented as $s^n(0)$) then $0+x=x$ is an inductive theorem of E.

The next requirement of the inductionless induction method is the existence of a first-order axiomatization A of the minimal Herbrand model of the theory E. A model M is smaller than a model M' if for every formula φ, $M' \models \varphi$ implies $M \models \varphi$. Let us denote the minimal model of the theory E as M_E. A recursive set A of first-order universal formulas is an axiomatization of the minimal Herbrand model if and only if:

- $M_E \models A$
- For every Herbrand model M of E, $M \models A$ implies that M is equivalent to M_E.

In general the axiomatization A is incomplete. It captures only some but not all properties of the model M_E.

The final requirement of the approach is that both the theory E and the theorem T which we want to prove in the theory E by induction have to be purely universally quantified. Putting all the above together we can prove the following theorem, laying in the core of the approach.

Theorem 60. *If A is a first-order axiomatization of a purely universal theory, T is a purely universal formula, and $E \wedge T \wedge A$ is consistent then T is an inductive theorem of E.*

What we want to prove is that for any Herbrand model M, $M \models E$ implies $M \models T$. Since $E \wedge T \wedge A$ is consistent and purely universal according to the Herbrand theorem [24] there is a Herbrand model M_0 such that $M_0 \models E$, $M_0 \models T$, and $M_0 \models A$. According to the second part of the definition of A, M_0 is equivalent to M_E. Since $M_0 \models T$ we conclude that

$M_E \models T$. According to the definition of M_E for any Herbrand model M, $M \models E$ implies $M \models T$. $\Box$

The above theorem reduces inductive theorem proving in a theory E to consistency checking in first-order logic. For the given theory in order to apply the approach we have to address two questions: how to find an approximation A and how to check consistency of formulas.

The approach can use any first order logic theorem prover to prove consistency. Since the inductionless induction originates from the rewriting community, Knuth-Bendix completion is usually used as the underling theorem prover. First order logic theorem provers are semidecidable procedures for inconsistency checking. In other words if the given formula is inconsistent then the theorem prover will eventually prove it. If the given formula is consistent then there are two possible outcomes, either the theorem prover will prove it or it will not terminate. In the last case the question regarding the formula consistency remains open. Since inductionless induction requires consistency checks and relies on first order logic theorem provers it is co-semidecidable. In other words, if a given formula is not an inductive theorem of the theory E then it will be eventually proven. On the other hand for some inductive theorems the approach will provide a proof while for others it will not terminate.

Let us now consider various approaches to the construction of the first-order axiomatization of the minimal Herbrand model.

If the theory E contains an equality predicate and is equivalent to a finite convergent rewrite system (in other words decidable) then A can be chosen as $false \neq true$. The details can be found in [105, 54].

If the theory E is equivalent to a finite convergent rewrite system and C is a set of free constructors of the theory then A can be chosen as:

- $f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \Rightarrow x_1 = y_1 \wedge \ldots \wedge y_n$ where $f \in C$
- $f(x_1, \ldots, x_n) \neq g(y_1, \ldots, y_m)$ where $f, g \in C$ and $f \neq g$

The approximation A axiomatize properties of the free constructors. The details can be found in [65].

Some other approaches to the approximation construction rely on ground reducibility. A term is ground reducible if all its ground instances are reducible. Such approaches neither assume any set of free constructors nor any equality predicates. The details can be found in [74, 16].

The limitations of the inductionless induction can be described by the following quotes from the paper "History and Future of Implicit and Inductionless Induction: Beware the Old Jade and the Zombie!" [147].

- "Inductionless induction has shown to be practically useless, mainly due to too many superfluous inferences, typically infinite runs, and too restrictive admissibility conditions.

- "There was the general opinion that inductionless induction was dead."
- "Do not to fumble around with the zombie of inductionless induction! The experts in implicit induction have spent a lot of time with it, mostly to bury it."

If the inductionless induction works for a given theory and a given theorem then we can prove the theorem without a user-provided induction scheme. On the other hand if there is no proper approximation of the Herbrand model or we just do not know it then the method cannot be used. If we have a proper approximation for the theory we receive a co-semidecidable procedure, which means that even if the given formula is a valid inductive theorem the method can fail to prove it.

On the other hand the stereotype based approach requires more user interaction but is more reliable. User interaction is mostly needed for the construction of stereotypes and universal transformations. As soon as we have stereotypes we reduce the verification problem to first order logic theorem proving and thus receive a semidecidable procedure.

7.4 Approaches based on interactive theorem proving

The approaches considered in this section do not prove the correctness of a program, but check the proofs provided by a user. Therefore these approaches can handle heap properties of arbitrary complexity.

The stereotype based verification is less expressive than the approaches considered in this section, but proposes better proof automatization and can be used by less qualified software developers.

Most of the interactive verification techniques use general purpose proof assistants such as PVS [116], Isabelle [113], or Coq [15]. Each of the above proof assistants provides a framework to deal with the verification of imperative programs. [129] describes a framework built on top of Isabelle/HOL for the verification of imperative programs. The framework includes a programming logic model and Hoare logic rules for both partial and total correctness. Another example of such a framework is Ynot which is described in [107, 108]. Ynot is an an extension of Coq to deal with separation logic. [142] describes an extension of the HOL4 interactive theorem prover to deal with separation logic.

Proof assistants were used to verify various complicated properties. For instance, [17] presents a formal verification of a C compiler front-end in Coq. Another example is the formalization and verification of the priority ceiling protocol in PVS, which is described in [93]. The priority ceiling protocol is a deadlock free extension of the priority inheritance protocol, which we verified with stereotypes.

Another approach to interactive verification is to develop a specialized interactive theorem prover, targeted on software verification. An example of this approach is the KeY tool which is described in [9]. KeY is an interactive theorem prover based verification tool targeted on object-oriented programs. The tools is based on dynamic logic [60]. The main difference between dynamic logic and Hoare logic is that dynamic logic can use programs instead of formulas to specify a program state. For instance, a while loop can be used to specify that a data structure is acyclic. To deal with such specifications the approach relies on user-provided inductive proofs. [130] demonstrates how dynamic logic can be combined with dynamic frames. The extension adds modifies clauses to the method specifications and describes a new logic rule which uses them.

7.5 Mixed approaches

In this subsection we consider tools which mix several of above approaches. The motivation for this a mix is to combine the expressivity of the interactive proof assistants with the ability to work without the help of a user of the less expressive tools. The simple problems can be verified by an automatic tool, and more complicated problems can be still be verified by means of user-provided proofs.

In spite of improved automatization there are still cases which require user-provided proofs. Therefore the approach still has the disadvantages of the pure interactive approaches.

[150] describes Jahob; a verification tool which combines the following tools:

- Decidable theories:
  - BAPA [89] a theorem prover for boolean algebra with Presburger arithmetic to deal with set cardinalities.
  - MONA [63] a theorem prover for monadic second-order logic of strings.
- **SMT** theorem provers:
  - CVC3 [8]
  - Z3 [39]
- **FOL** theorem provers:
  - SPASS [144]
  - E [131]
- Interactive theorem provers:

  - Coq [15]
  - Isabelle [113]

The main idea of the approach is to combine all of the above tools in a sound but not complete way. The integration is based on the concept of formula approximation, which maps an arbitrary formula into a semantically stronger but simpler formula. The approximated formula can be split into disjoint formulas each of which can be verified by one of the above tools. Since the resulting formula is stronger, the approach is sound.

The tool was used to verify the following examples: hash table, binary search tree, array list, priority queue, circular list, singly-linked list.

Another tool which uses a mixed approach is VCC [31]. The tool targets verification of concurrent C. VCC combines an ownership methodology with interactive theorem proving. To handle user-provided proofs VCC uses HOL-Boogie [18]. HOL-Boogie is an interactive proof-environment which combines Boogie and Isabel. Among other things the environment can be used to proof lemmas about data-types and memory management. An application of VCC to Microsoft's Hyper-V Hypervisor is described in [93]. The Hypervisor is a software which runs on a single multi-processor machine several virtual multi-processor machines. The size of the Hypervisor is 100 KLOC of C and 5 KLOC of assembly, respectively.

Chapter 8

Conclusion

This chapter is organized as follows. First we summarize the stereotype based verification methodology and discuss the contributions. Then we briefly sketch avenues for future development and evaluation of the methodology. In the last section we discuss how various areas of software verification, software engineering, and theorem proving could benefit from the presented methodology.

8.1 Summary and contributions

Initially our ultimate goal was to develop a complete methodology which enables the specification and semiautomatic verification of heap structures and semantical relations in automatic theorem provers. Here we have to clarify several things:

- by a complete methodology we mean that any heap structure, semantical relation, and operation can be covered by the methodology. We believe that this requirement is crucial for the following reason. Verification has a holistic nature. If a small part of a system cannot be verified then the whole system cannot be verified. On the other hand, if the verified system is big enough then eventually we will run into all kinds of exotic properties and operations, which then prevents the verification of the whole system.
- another requirement which we impose on the methodology is the precise description of the steps which have to be done in order to specify and verify any given example. This property has to eliminate guessing from the approach. At the very beginning of the specification/verification process clients have to know all decisions they have to make and problems they have to address.
- our experience with automatic theorem provers suggests that additional hints are unavoidable. Nevertheless, it is possible to reduce

significantly the complexity of these hints. We would like to keep these hints as simple as possible. In the first place we would like to avoid inductive proofs, quantifier alternations, and introducing extra functional symbols. In our approach we reduce hints to annotations with the universally quantified assertions. We call a verification process which involves only such hints semiautomatic. It requires some hints from a user, but these hints are quite straightforward and usually can be provided easily.

We already have identified another crucial methodology requirement. Because of the enormous specification complexity, we have to take care of reusability. We believe that the specification complexity reflects limitations of state-of-the-art automatic theorem provers. There is no evidence that these limitations will be resolved in the foreseeable future. All of the above suggests that dealing with enormous complexity is unavoidable, which means that reusability is a must for a heap verification methodology.

The rest of the section is organized in the following way. We describe various aspects and mechanisms of the stereotype methodology and bind them with the methodology requirements which they address.

- **Stereotype:**
  We began the development of our methodology from the identification of entities which are essential for the specification and verification of any heap structure or semantical relation. We try to keep them as general as possible:

  - **Stereotype items** are values of interest of the relation.
  - **Stereotype invariants** are properties of stereotype items.
  - **Stereotype operations** are updates of stereotype items which preserve stereotype invariants.

  Since these entities are inseparable, we couple them into one entity, a **stereotype**. By this we achieve a high reusability. Once a stereotype is developed it can be easily reused. By reusing a stereotype we reuse values of interest, properties, operations, and approximations (e.g., how we approximate a transitive closure by a stereotype item). On the other hand, using stereotypes we get a completeness. Any structure can be described precisely enough by introducing extra stereotype items and extra invariants. Extra invariants free clients from the necessity to prove inductive properties. The price which we have to pay for this completeness is extra complexity, for the development of stereotype operations. They have to update all extra stereotype items and preserve all extra stereotype invariants.

- **Stereotype dynamic model:**
  When we deal with a heap data structure it is typical that we have to deal not only with a single instance but with a set of instances. This complicates both proof obligations which have to be verified by an automatic theorem prover and specifications which have to be provided by a client. To address this issue we introduce stereotype slices and stereotype instances. A stereotype slice is an instance of a stereotype. A stereotype slice adds stereotype items to the definitions of classes which participate in the stereotype slice. While a stereotype slice covers the whole heap, a stereotype instance corresponds to a single data structure or a design pattern.

  The distinction between stereotype slices and instances has several advantages:

  - The property which guarantees disjointness of stereotype instances is reused together with a stereotype. Since the property can be used to specify disjointness between different instances of a data structure or a design patterns, its reusability reduces specification overhead as well as facilitates automatic verification.
  - A stereotype instance is a natural unit for the specification of frames of stereotype operations. We use sets of stereotype items to describe a cumulative effect of inductive composite stereotype operations and of a parallel composition of stereotype operations.
  - The distinction between stereotype slices and instances is used in our methodology to guarantee the preservation of stereotype invariants by stereotype operations.

  Overall, a stereotype slice indicates a semantic relation between classes, and a stereotype instance indicates a semantic relation between instances of these classes.

- **A language for the description stereotype operations:**

  The expressiveness of stereotype operations is crucial for the successful application of the stereotype approach. Preferably we would like to be able to specify any stereotype operation for any stereotype. We address this issue by introducing the specification language of specifications (**SLS**). **SLS** provides a natural operational way for developing stereotype operations. A developer of a specific stereotype can use **SLS** to specify a stereotype universal transformation: a transformation which can be used to specify an arbitrary stereotype operation. To demonstrate how this approach works we used **SLS** to specify universal transformations for `Sequence` and `Tree` stereotypes.

SLS is well suited for human specification writers, but hardly usable by an automatic theorem prover. **SLS** is both operational and recursive, but the automatic theorem provers are targeted towards dealing with functional and non-recursive specifications. We provide an automatic translation from an **SLS** term to a standard stereotype operation representation (**SSOR**) to overcome this mismatch. Essentially an **SSOR** is a pair of universally quantified pre- and post-conditions. An **SSOR** is both functional and non-recursive. It is perfectly feasible for an automatic theorem prover.

By using **SLS** together with the automatic translation into **SSOR** we achieve the complete expressiveness of stereotype operations.

- **Coupling between the stereotype slices and the heap:**

  As soon as the development of a stereotype has been completed it can be used for source code specification and verification. For this we use glue invariants and glue invariant states. A glue invariant state describes the role which an object plays in a topological structure or in a semantics relation. A glue invariant is a universally quantified formula which couples together stereotype items, heap fields, and glue invariant states. By introducing glue invariants we accomplish complete independence of stereotypes from the heap. Because of this independence stereotypes are more reusable. We can use the same stereotype to specify different data structures; e.g., we can use the sequence stereotype to verify a singly-linked list, a doubly-linked list, a cyclic list, a path in a tree, etc. We achieve the flexibility by moving the part of specifications which is specific to the verified program into the glue invariant.

  Another advantage of glue invariants is information hiding. The stereotype invariant states, together with the stereotype items, are expressive enough to specify the source code without mentioning heap fields, arrays, and containers which are used in the implementation. The client can reason about a relation between an object which participates in a data structure or design pattern and the rest of the objects in terms of the stereotype items and the glue invariant state of the object. In this way both precision and abstraction of specifications can be achieved.

  Another aspect of the integration of stereotypes with the source code is ghost updates and stereotype specifications. For both of them we use the same stereotype operations. Since stereotype updates can be extremely verbose and complicated we prevent an extensive specification duplication by the dual application of stereotype operations.

We proposed the first *complete* methodology for the specification and verification of heap structures and semantic relations with automatic theorem provers. We do not claim in any way that we closed the problem and proposed a perfect solution. A significant number of technical and theoretical issues has to be resolved before the proposed technique can be used to specify and verify full-scale real life industrial projects. We sketch these issues in the next section. Even though the proposed methodology uses automatic theorem provers it is far from being automatic. A significant amount of work has to be done to verify a program, which results in both technical and social complications. A development becomes much more expensive, requires more time, more experience, and educated developers. Nevertheless, we believe that additional work is unavoidable due to the state of the art of automatic theorem proving. We also believe that after resolving this issue it would be possible to apply the provided technique in industrial projects.

8.2 Directions for future work

As we have mentioned in the previous section quite a few issues have to be resolved before we are able to apply the stereotype based methodology in industrial projects. In this section we briefly describe those of them which are observed even without a full-scale case study. It is possible that a case study will identify even more theoretical and technical issues. For now let us consider the known issues:

- **Development of a standard stereotypes library:** Since the stereotype methodology makes a strong impact on reusability, it makes sense to provide a standard library of stereotypes. The library has to contain typical stereotypes equipped with universal transformations. This library could provide a solid foundation for general purpose program verification. Hopefully, the standard library could do the hardest part of the job: to provide a precise enough approximation of the specified relation and operations for their update. The job of a library's client is to adjust them by means of glue invariants to specify the desired source code properties. Another advantage of a standard library is a specification standardization and improved readability.

  For now the standard stereotype library consists of relation inversion, sequence, and tree stereotypes. There are two obvious directions for the standard library development.

  - The first direction is the addition of new stereotypes to the library. For instance, it would be highly desirable to have DAG and graph stereotypes in the library. Nevertheless, there are less obvious candidates for inclusion in the standard library. For instance, most of 3D CAD systems work with triangulated surfaces. The data structure which represents a triangulated surface has more specific properties than a general graph. This class of examples motivates the addition of a planar graph stereotype to the library.

  The biggest challenge in the standard library development is the development of universal transformations. Each stereotype has its own distinguished properties which have to be appreciated by the universal transformation. For instance, a DAG has data sharing and a graph has loops of irregular structure. It is difficult to estimate whether it will be easy or not to develop universal transformations which appreciate these properties. The only way to check this is to try to develop these universal transformations.

  - Another direction for the standard library development is to improve the precision of the stereotypes' approximation. This can be done by adding new stereotype items and invariants. For instance, in the sequence and tree stereotypes we consciously ignored all properties related to natural numbers; e.g., properties of the size of a sequence, of indexed element access in a sequence, of access of a tree element by a sequence of indexes, and so on. These properties are redundant and can be constructed from current stereotype items. Nevertheless, these definitions will be recursive and their proof requires induction.

    In order to improve the precision of the approximation of stereotypes we have to add both stereotype items and stereotype invariants to the stereotype. For instance, to add the index of an element to the sequence stereotype we have to add the index stereotype items and several invariants: index of the next element is greater by one, all elements in the right tail have greater value, the index of the last element is equal to the size of the sequence, and so on.

- **Stereotype specification inference:**

  In the stereotype methodology a single heap update is duplicated in several different ways:

  - as a class fields update;
  - as a glue invariant states update;
  - as a stereotype items update;
  - and as a method specification.

  These duplications increase the overhead for a methodology user. The negative impact of the duplications can be decreased by inference. For now we can see the following opportunities for inference:

- Class fields and glue invariant state updates can be used to infer stereotype ghost updates. For instance, if a glue invariant states that a field is always equal to the `next` stereotype item and if we assign a new value to the field then we can infer a pair of stereotype operation calls. One of them removes the old relation and one adds the new relation. Another example is, if we change a glue invariant state of an object from a state in which it participates in a tree to a state in which it participates in a sequence then we can infer a pair of stereotype operations call one of which removes the tree relation and the other one adds the sequence relation.

  To make this inference feasible we have to syntactically distinguish glue invariants which establish the equality between a heap field and a stereotype item for a given glue invariant state. Usually such invariants look like $st = S \Rightarrow it = f$, where st is a ghost variable which represents the glue invariant state, S is a concrete value of the glue invariant state, it is a stereotype item, and f is a class field. To handle temporal invariant violations one would infer ghost updates not for a separate field update but for the whole code fragment which restores the glue invariant.

- Stereotype ghost updates can be used to infer stereotype method specifications. Since both of them use **SLS** terms, one way to infer stereotype method specifications is to collect all stereotype method calls and replace normal method calls with their stereotype specifications. The resulting sliced method will consist only of stereotype operations and can be used as the method specification. This approach looks pretty straightforward, nevertheless it has several essential drawbacks:

  * The approach would not work for recursive methods. For recursive methods, specifications play a role similar to the one loop invariants play in a loop. A post-condition of a recursive method has to aggregate updates which are done by all nested calls. Inferring this specification requires automatic contraction of an aggregated specification, which is hardly feasible in the general case.
  * A method specification can contain stereotype specifications not in a form of a **SLS** expression, but in a form of explicit logical expressions. Since explicit logical expressions are not a part of **SLS** we cannot integrate them into the rest of the inferred method specification. It is even possible that this expression cannot be represented as an **SLS** term at all. At the end we cannot infer stereotype method specification for a method whose body contains a call of a method with a

non-**SLS** stereotype specification.

- The input parameters of a stereotype operation which is called from a method body are side effect free expressions. It is possible that these expressions mention a heap value or a local variable. The method body specification can mention only heap states before and after the method execution and local values before the method execution. We cannot express the stereotype operation parameters in the method specifications if the stereotype oration is executed in the middle of the method body. The parameters are reconstructed by the weakest pre-condition calculus which introduces explicit values to represent intermediate heaps. It is not acceptable to introduce these variables in the method specifications.

Because of these drawbacks we cannot infer method stereotype specifications in the general case, but only in simple cases. Nevertheless, it still can be useful since it can relieve the client from specifying many simple methods.

- **Tool support:**

  For now the stereotype based verification is a theoretical technique. To bring it to practice we have to provide a tool support. More specifically we need an implementation of a simplistic stereotype based verification language which supports the following features:

  - A stereotype definition:
    - stereotype items.
    - stereotype invariants.
    - **SLS** support for stereotype operations.
  - Stereotype and source code integration facilities:
    - stereotype slices and participation of classes in slices.
    - glue invariant states.
    - glue invariants.
    - ghost stereotype updates via stereotype operations calls from the source code.
  - Behavioral specifications:
    - method pre- and post-conditions.
    - behavioral invariant states.
    - behavioral invariants.

  A natural way to implement this language is to define a translation into **Boogie**. The thesis covers all aspects of this translation.

Another feature which is crucial for the successful application of the approach is a more efficient usage of hints. As it has already been mentioned we had to add hints to enable verification of some programs and universal transformations. Such hints are introduced as intermediate assertions. For instance, if an **SMT** theorem prover cannot prove $\{\phi\}S;Q\{\phi'\}$ we could introduce this intermediate hypothesis φ such that both $\{\phi\}S\{\varphi\}$ and $\{\varphi\}Q\{\phi'\}$ hold. The existence of φ in the obvious way proves $\{\phi\}S;Q\{\phi'\}$. Another type of hint which we used is a case split. If an **SMT** theorem prover cannot prove $\{\phi\}S\{\phi'\}$ then we introduce formulas $\varphi_1, \ldots, \varphi_n$ such that $\bigvee_{i=1}^{n} \varphi_i$ and $\{\varphi_i \wedge \phi\}S\{\phi'\}$ for all $i \in [1..n]$ hold. The existence of $\varphi_1, \ldots, \varphi_n$ implies $\{\phi\}S\{\phi'\}$. For now, we introduce these hints as ordinary **Boogie** assertions. Instead of this the weakest pre-condition calculus used by **Boogie** can be modified to utilize the hints in a more efficient way. We believe that due to the current state of **SMT** theorem provers these hints are unavoidable and have to be adequately handled and utilized by verification languages.

- **Evaluation:**

  An important way to evaluate the approach presented in this thesis is to make case studies. Since we did not have a prototype implementation we used a direct encoding into **Boogie** to perform a limited evaluation. We can distinguish the following types of case studies which can be performed to evaluate this approach further:

  - **Evaluation of the SLS-to-SSOR transformation.** The transformation is used for the verification of stereotype operations. A natural case study for the transformation is the verification of universal stereotype transformations. As we have mentioned above we did not have a tool and used the weakest precondition calculus provided by **Boogie** as a replacement. Even without the **SLS**-to-**SSOR** transformation we were able to verify the sequence universal transformation and some parts of the tree universal transformation. These examples showed that universal transformation verification can be quite challenging. We believe that the **SLS**-to-**SSOR** transformation could improve the situation. The hypothesis can be practically evaluated only after the prototype has been implemented.
  - **Verification of data structures and design patterns.** Using universal sequence transformation and partially implemented tree universal transformation we were able to verify several data structures and design patterns. The provided experiments demonstrate that the approach works for some challenging examples. Nevertheless, they are obviously far from being complete. To

achieve completeness we can use two sources of examples: a standard design patterns text books (e.g. [50]) and data structure libraries (e.g., the Java collections framework or C++ STL). Such case studies would be a enabled by a prototype implementation.

- **Evaluation of a full-scale industrial project.** At the end of the day the only way to evaluate a verification technique is a real life industrial project. Nevertheless, it does not make a lot of sense to approach such an ambitions project before the obvious problems are addressed. That is why we believe that this case study will be crucial, but should be done only after all the above items have been be addressed.

8.3 Applications and consequences

In this section we briefly sketch the possible applications of the proposed methodology. Since the problem of verifying heap topologies and semantical relations has been researched for a long time, some of these applications are well known. On the other hand, some applications are specific for the proposed methodology. For a clear picture we would like to mention all of them.

- **Verification of multiple objects invariants.** A multiple objects invariant is a property relates multiple objects. The number of objects which participate in a multiple objects invariant is potentially unbounded. On the other hand the same object can participate in an unbounded number of invariants. For these reasons multiple objects invariants is challenging. To enable verification of the invariants we have to be able to specify and verify on which objects an invariant depends and in which invariants an object participates. The heap topology can be used to extract this information. We also need to be able to specify which objects are potentially affected by a method call. The stereotype based methodology can be used to address all these needs. Stereotype items can be used to specify invariants and stereotype operations provide a natural way for the specification of methods effects. In our experiments we used a simple visible state semantics for invariants. The visible state semantics was chosen to avoid extra complexity. On the other hand, the stereotype based methodology can be used as a basis for any other invariant verification technique. In this way an invariant verification technique can reuse heap specification facilities provided by stereotypes and concentrate on the description of which invariants can be assumed and asserted at which program points.

- **Verification of concurrent programs.** The stereotype based methodology can be used as a bases for the verification of the following concurrent properties:

- **Absence of race conditions.** A race condition arises when two or more threads in a concurrent way modify the same part of the heap. One of the properties which have to be checked to prevent race conditions is the disjointness of thread effects. Again stereotype items and operations can be used to specify which objects could be affected by an execution of a thread in a given state. The absence of race conditions can be verified by checking the pairwise disjointness of the effects of the concurrently executed threads. The effect of a thread is defined by the frame of the thread body.
- **Deadlock freedom.** A deadlock arises when two or more threads lock the same objects in a different order. To prevent a deadlock it is enough to introduce a partial order on locked objects, and check that each thread locks them according to this order. Quite often such an order is retrieved from an acyclic data structure, e.g., a list or a tree path. For instance, a linked list can be modified in a deadlock-free way by an unbounded number of threads if each of them locks and releases object according to the order induced by the "next element" relation of the list. On the other hand, if a loop is created in the list then the relation induced by the "next element" is not an order any more (the antisymmetry of the order is violated). In this case a deadlock can arise even if elements are locked according to the order. A way to prevent this is to verify that a cycle is never created in the list. Stereotypes provide a natural way to specify such properties. For instance, the sequence stereotype can be used to verify that this property holds for a linked list. In a similar way other stereotypes can be used to verify that a given part of a heap structure is acyclic and as a result induces a partial order which can be used for deadlock prevention.

- **Software engineering applications.** At the current state software engineering is a quite fuzzy area of engineering. Therefore, the development and maintenance of software systems is extremely challenging, error prone, and hardly manageable. For now software engineering is more an art than a science. We believe that in the long run the stereotype based methodology could improve this situation. This assumption is based on the following observations:
  - A significant part of the complexity of software engineering comes from dealing with semantical relations and heap topology.

– The stereotype based methodology proposes a way to address these issues.

Let us consider several possible applications of the stereotype methodology in the area of software engineering:

– **Encapsulation and modular development.** A strongly desired property for a software engineering project is modularity. It should be possible to split the whole project into independent modules in such a way that all communication between modules is going strictly via predefined module interfaces. The whole project could be organized as a hierarchical composition of modules (a module can contain other modules). This project organization has the following advantages:

  ∗ *Independent development.* Each module can be developed and tested in isolation from other modules. It enables parallel and distributed development.

  ∗ *Interchangeability.* An implementation of a module can be changed without affecting the rest of the project. The new version of the module has to respect the interface but can freely change the implementation.

  ∗ *Manageability.* When a module has been developed we need to take into consideration only those modules which are directly accessible from the developed module and can ignore the rest of the system.

  Unfortunately it is imposable to achieve true modularity with modern programming languages. The reason is aliasing. There is no mechanism which prevents an undesired aliasing from one module to the internals of another one. Typically it happens by means of alias leaking (a method returns a reference to an internal of the module) and capturing (a methods saves an input reference into the module's internals).

  To prevent alias leaking from a module and alias capturing by a module we have to be able to detect aliasing at compile time. Stereotypes can be used to describe heap relations inside the module. Later on this specification can be used to prevent undesired aliasing and ensure encapsulation of the module's internals.

– **Design patterns.** A Design pattern is a reusable solution of a typical problem in software engineering. Since design patterns provide various benefits they are widely used in everyday software development. Despite the prevalence there is still no direct support for design patterns in modern programming languages. The absence of direct support results in misuses of design patterns, worse code readability, and code duplication. Defacto there is even no generally accepted formal definition of design pattern.

We believe that the main reason for such a fuzzy state of design patterns is the absence of a mechanism for the specification of semantical relations in modern programming languages. A typical design pattern includes a description of:

  ∗ roles of participating objects. (e.g. Observer, Iterator, Adapter, and so on). These includes descriptions of methods which have to be provided by an object which plays a given role.
  ∗ semantic relations between participating objects (observer observes a specific subject, iterator iterates over a specific container, adapter owns a specific adoptee)
  ∗ description of an order and conditions under which methods provided by participating objects can be executed.

The first item is usually provided by interfaces and abstract classes. If a role has a partially predefined behavior which can be reused then an abstract class can be used to represent the role. Otherwise a role is represented by an interface. A good example of role descriptions in modern programming languages is the observer pattern implementation in the **Java** standard library. Since subject has a standard behavior it is represented by an abstract class. On the other hand, observer is represented by an interface because it does not have a standard behavior.

The other two items are not really provided by modern programming languages. For instance, by looking at the signature of **Java**'s "Observable" we can assume that there is some relation between it and the "Observer" interface, but we have no idea which kind of relation it is and how different observers and observables communicate with each other.

This gap can be covered by the stereotype based methodology.
Stereotypes can be used to describe semantic relations between objects which play various roles. For instance, we used the relation inversion stereotype to describe this semantic relation for the observer, iterator, and mediator patterns. As soon as we have a semantic relation between objects which participate in a design pattern we can use them to specify pre-conditions, post-conditions and invariants. These specifications precisely define which methods can be executed, and in which order.

- **Verification of security properties.** The stereotype based methodology can be used to verify some security properties. One of these properties is isolation. The isolation property guarantees that a given piece of the code would not affect the rest of the system. For instance, an operating system enforces isolation between different processes. In this way a corrupted process with low access rights cannot affect a process with high access rights. Another example is process isolation in web browsers. For instance, if we are browsing two web pages and enter a credit card number in one of them we would like to be sure that this information will be isolated from the other one. Isolation is also strongly desirable for untrusted bytecode (typically downloaded from internet).

In all of the above situations stereotypes can be used to guarantee isolation. For brevity we refer to pieces of code whose isolation is checked as process. The isolation can be verified in the following way:

- a topology of all processes for which isolation has to be checked is described by means of the stereotype based methodology. If for any process there is only an untrusted bytecode but no source code then Proof-Carrying Code has to be used [110]
- As soon as the topology of a process is described we can use stereotype items to specify effects of the process.
- The last thing which we have to do is to check pairwise disjointness of effects of participating processes.

The stereotype based methodology could be even more useful when a limited interprocess communication between isolated process is provided. For instance, this communication is provided by most operating systems. A transfer of a piece of memory from an isolated process p_{old} to another isolated process p_{new} is typically done by copying the piece of memory. The memory copying guarantees process isolation, but requires extra cpu time and memory. Another way to address this issue is to prove at compile time that the transferred piece of memory will not be accessed by the process p_{old}. For instance, this approach is used by Singularity OS [61]. Singularity OS exploits a type system to guarantee the isolation property. The limitations of this approach originate from the limitations of the exploited types system; the transferred pieces of memory can be aliased only in a very specific way. The stereotype based methodology provides a way to overcome these limitations. Stereotypes can be used to describe an arbitrary topology. During a compilation the stereotype specifications can be used to verify that the transferred piece of memory will not be accessed by the process p_{old}.

- **A foundation for specialized automatic theorem provers.**

  In general automatic theorem provers try to solve an undecidable problem. So it is not a surprise that their abilities are limited. Even a very simple formula can have a huge proof. A good example is Fermat's last theorem. It is formulated in a single line, but its proof [145, 146]:

- took 358 years
- was done using very advanced mathematical concepts
- is more then two hundred pages long

On the other hand it was proven that the proof of Fermat's last theorem can be done in elementary arithmetic [2]. Elementary arithmetic is a relatively simple theory which consists of increment, addition, multiplication, power, and the induction schema for bounded formulas.

The example demonstrates the delusive nature of mathematical logic. A proof of a logical statement which looks very simple and provable in a simple theory can be fantastically complicated. For this reason, we cannot expect that automatic theorem provers will be able to prove an arbitrary logical statement. For now, even people cannot do it.

From these arguments we can conclude that if we develop an automatic theorem prover we have to target it towards a very specific set of formulas. The set has to be both rich enough to express most of the desired properties and simple enough to be feasible for automatic verification. It is hard to identify a logical characterization of this set, so even an extensive set of examples and cases studies can be of great use.

We believe that the stereotype based methodology can be used to identify this set of formulas for a theory of heap structures and semantic relations. Stereotype invariants identify typical properties which describe a state of a heap manipulating program. Stereotype operations identify typical relations between two program states. User provided assertions (hints) identify transactions between program states which are especially hard for automatic verification.

Appendix A

Universal transformations for sequence stereotype

In this appendix we construct the universal transformations for the sequence stereotype. As have we mentioned in **Section 6.5** it is enough to construct the universal set of relation addition and removal to get a universal transformation.

A.1 Operation `addSetSequenceRelation`

The stereotype operation `addSetSequenceRelation` merges a set of sequence instances into a single instance. The operation definition is shown in Figure A.1.

The input parameters of the operation are the following:

- Ib is the instance identifier of the first sequence.
- Ie is the instance identifier of the last sequence.
- $Inst$ is the set of instance identifiers of the merged sequences.
- $nextInst$ is an order over sequences. It maps a sequence identifier into a sequence identifier of the next sequence according to the order in which they are merged.
- $ElUnion$ is a redundant argument which maps an instance identifier o into a union of all elements of all sequences which are succeeded or equal to o. The parameter is redundant in a sense that it can be constructed from other parameters, but we require it to avoid the usage of transitive closure and to facilitate the proof.

The operation has three local variables. $nextInst^*$ is the transitive closure of $nextInst$. We use the redundant parameter $ElUnion$ to define it.

```
addSetSequenceRelation⟨Sequence⟩($Ib : ref!, Ie : ref!, Inst : Reg!,$
    $nextInst : ref \to ref, ElUnion : ref \to TReg)$
Local variables:
    $\forall o : nextInst^*[o] = ElUnion[o] \cap Inst\,;$
    $Inst' = Inst \setminus \{Ib\}\,;$
    $Ib' = nextInst[Ib]\,;$
Pre-conditions:
    $\forall o \in Inst : o.\mathtt{instID} = o\,;$
    $Ib \in Inst\,;$
    $Ie \in Inst\,;$
    $nextInst(Inst \setminus \{Ie\}) \subseteq Inst\,;$

    $ElUnion[Ie] = Ie.Elements\,;$
    $\forall o \in Inst \setminus \{Ie\} : ElUnion[o] = ElUnion[nextInst[o]] \cup o.Elements\,;$
    $\forall o \in Inst \setminus \{Ie\} : o.Elements \sharp ElUnion[nextInst[o]]\,;$
    $\forall o \in Inst, o' \in Inst : o' \in ElUnion[o] \Rightarrow ElUnion[o'] \subseteq ElUnion[o]\,;$

    $nextInst^*[Ie] = \{Ie\}\,;$
    $\forall o \in Inst \setminus \{Ie\} : nextInst^*[o] = \{o\} \cup nextInst^*[nextInst[o]]\,;$

    $Inst \subseteq ElUnion[Ib]\,;$
    $\forall o, o' \in Inst \setminus \{Ie\} :$
        $o' \in ElUnion[o] \Rightarrow nextInst[o'] \in ElUnion[nextInst[o]]\,;$

Transformations:
    $\langle ElUnion[Ib], Elements, ElUnion[Ib] \rangle\,;$
    $\langle ElUnion[Ib], \mathtt{first}, Ib.\mathtt{first} \rangle\,;$
    $\langle ElUnion[Ib], \mathtt{last}, Ie.\mathtt{last} \rangle\,;$
    $\langle Inst \setminus \{Ie\})\langle o' \rangle.\mathtt{last}, \mathtt{next}, nextInst[o'].\mathtt{first} \rangle\,;$
    $\langle \exists o, o' : o \in Inst' \wedge o' \in Inst' \wedge o = nextInst[o']$
        $\wedge \mathbf{v} = o.\mathtt{first}, \mathtt{prev}, o'.\mathtt{last} \rangle\,;$
    $\langle (Inst \setminus \{Ie\})\langle o' \rangle.Elements, \mathtt{Next}^*, \mathbf{v}.\mathtt{Next}^* \cup ElUnion[nextInst[o']] \rangle\,;$
    $\langle (Inst \setminus \{Ib\})\langle o' \rangle.Elements, \mathtt{Prev}^*,$
        $\mathbf{v}.\mathtt{Prev}^* \cup ElUnion[Ib] \setminus ElUnion[o'])\,;$
Input instances: $Inst$
Frame:
    $\langle \mathbf{v} \neq \mathbf{v}.\mathtt{last} \vee \mathbf{v} = Ie.\mathtt{last}, \mathtt{next} \rangle\,;$
    $\langle \mathbf{v} \neq \mathbf{v}.\mathtt{first} \vee \mathbf{v} = Ib.\mathtt{first}, \mathtt{prev} \rangle\,;$
    $\langle Ie.Elements, \mathtt{Next}^* \rangle\,;$
    $\langle Ib.Elements, \mathtt{Prev}^* \rangle\,;$
Measure: $Inst$
{
    if $(Ib \neq Ie)$ then {
      addSetSequenceRelation($Ib', Ie, Inst', nextInst, ElUnion$)
          measure $Inst' \overset{Ib}{\subset} Inst\,;$
      addSequenceRelation($Ib, Ib'$);
    }
}
```

Figure A.1: `addSetSequenceRelation` stereotype operation.

$Inst'$ and Ib' are corresponding parameters of the recursive call. We construct $Inst'$ by dropping Ib from $Inst$. Ib' is defined as the identifier of the sequence which succeeds the first one.

The pre-condition section of `addSetSequenceRelation` consists of several subsections. In the first one we define the properties of domains of the input parameters. $Inst$ contains only the instance identifiers, Ib and Ie are the elements of $Inst$, and the image of all elements of $Inst$ excluding Ie under $nextInst$ also belongs to $Inst$. The last property guarantees that $nextInst$ maps an instance identifier into another instance identifier.

The next subsection defines the properties of $ElUnion$. $ElUnion$ of the last sequence is equal to the elements of the last sequence. $ElUnion$ of all other sequences are equal to the elements of the sequence plus $ElUnion$ of the next sequence. The next property states the disjointness of the elements of a sequence and $ElUnion$ of the next sequence. This property guarantees the acyclicity of the constructed sequence. The last property of this subsection states that $ElUnion$ forms a sequence of the nested sets. If an element of $Inst$ belongs to $ElUnion$ of another element of $Inst$ then $ElUnion$ of the first element is a subset of $ElUnion$ of the second element. This property is redundant, nevertheless we include it as the explicit pre-condition since its proof requires the use of induction. We can think of this property as the inductive hypothesis.

The next subsection defines the semantics of $nextInst^*$. $nextInst^*$ of the last sequence is the singleton set $\{Ie\}$. $nextInst^*$ of all other sequences is equal to the elements plus $nextInst^*$ of the next sequence.

The last subsection contains two properties. The first one states that $Inst$ is a subset of $ElUnion$ of the first sequence. We need this property to verify the frame rules. The last pre-condition states that if an element belongs to the $ElUnion$ of another element then $nextInst$ of the first element belongs to the $ElUnion$ of the $nextInst$ of the second element. This property is another example of the redundant inductive hypothesis.

Let us now look at the transformation section of `addSetSequenceRelation`. Here we have one transformation per stereotype item.

- *Elements*: We know that $ElUnion(Ib)$ is equal to the union of all elements of all sequences. We set *Elements* of all object from $ElUnion(Ib)$ to $ElUnion(Ib)$.
- `first`: The first element of the resulting sequence is equal to the first element of the first sequence.
- `last`: The last element of the resulting sequence is equal to the last element of the last sequence.
- `next`: We change the value of `next` of the last elements of all sequences excluding the last one. o' denotes the instance identifier of

the affected sequence. The new value of `next` is the first element of the next sequence. By this transformation we establish a forward relation between sequences.

- `prev`: We change the value of `prev` of the first elements of all sequences excluding the first one. o denotes the instance identifier of the affected sequence. The new value of `prev` is the last element of the previous sequence. Since we do not have an explicit relation inversion we simulate it using the quantified variable o'. The transformation rule is equivalent to $\langle (Inst \setminus \{Ib\})\langle o'\rangle.\texttt{first}, \texttt{prev}, nextInst^{-1}[o'].\texttt{last}\rangle$, where $nextInst^{-1}$ denotes the inversion of $nextInst$. By this transformation we establish a backward relation between sequences.

- `Next`*: We change the values of `Next`* for elements of all sequences excluding the last one. o' denotes the instance identifier of an affected sequence. The value of `Next`* is increased by the addition of the union of the elements of all sequences which succeed o'. This union is equal to *ElUnion* of the next sequence.

- `Prev`*: We change the values of `Prev`* for the elements of all sequences excluding the first one. o' denotes the instance identifier of an affected sequence. The value of `Prev`* is increased by addition of the union of the elements of all sequences which precede o'. This union is equal to the difference between *ElUnion* of the first sequence and *ElUnion* of o'.

The input instance section states that the instance identifiers of all affected instances belong to $Inst$.

The structure of the frame section is similar to the transformation section. We have a frame rule per stereotype item:

- `next`: The operation preserves the values of `next` for all objects which are not a last element of a sequence and for the last element of the last sequence.

- `prev`: The operation preserves the values of `prev` for all objects which are not a first element of a sequence and for the first element of the first sequence.

- `Next`*: The values of `Next`* are preserved for all elements of the last sequence.

- `Prev`*: The values of `Prev`* are preserved for all elements of the first sequence.

- *Elements*, `first`, and `last`: there are no frame rules for *Elements*, `first`, and `last` since they are automatically inferred from the transformation rules.

The measure section states that the input parameter $Inst$ is used as a measure variable, which means that for each recursive call the value of $Inst$ has to decrease.

The last section is the method body. If $Inst$ is a singleton set, which is the case if and only if $Ib = Ie$, then we already have a single sequence. In the other case we perform a recursive call which merges all sequences but the first one, and then call `addSequenceRelation` to merge the first sequence with the result of the recursive call. In the measure section of the call we explicitly state that Ib belongs to $Inst$ but not to $Inst'$. We use it as a hint to prove that $Inst' \subset Inst$.

A.2 Operation `removeSetSequenceRelation`

Let us now consider the `removeSetSequenceRelation` operation. The operation is an inversion of `addSetSequenceRelation`. It splits a single input sequence instance into several sequence instances. Because of the similarity between the two operations we consider `removeSetSequenceRelation` in less detail than `addSetSequenceRelation`.

The operation is defined in Figure A.2. The input parameters of the operation have the following meaning:

- $Inst$ is a set of cut-points. It contains the initial points of relations which are removed by the operation.
- $nextInst$ is an order on $Inst$ induced by the `next` relation of the sequence. In other words, for each o from $Inst \setminus \{Ie\}$, $nextInst[o]$ is reachable from o by `next` and there is no other element from $Inst$ between them.
- Ib and Ie are the first and the last cut-points according to the order provided by $nextInst$.

The operation has six local variables. $nextInst^*$, $Inst'$, and Ib' have the same meaning as the corresponding variables of `addSequenceRelation`. The only difference here is that we use `Next`* instead of $ElUnion$ to define $nextInst^*$. $ElFirst$ defines the elements of the first output sequence as all elements from the beginning of the input sequence to the first cut-point. $MiddleEl$ defines the elements of the middle output sequences as the elements between two adjacent cut-points. $ElFirst$ defines the elements of the last output sequence as all elements from the last cut-point to the end of the input sequence.

The pre-condition section of `removeSetSequenceRelation` consists of several subsections. In the first one we define the properties of the domains of the input parameters. Since Ie is the last cut-point there has to be a

```
removeSetSequenceRelation⟨Sequence⟩
    $(Ib : ref!, Ie : ref!, Inst : Reg!, nextInst : ref \to ref)$
Local variables:
    $\forall o : nextInst^*[o] = \texttt{Next}^*[o] \cap Inst\,;$
    $Inst' = Inst \setminus \{Ib\}\,;$
    $Ib' = nextInst[Ib]\,;$
    $ElFirst = Ib.Elements \setminus Ib.\texttt{Next}^*\,;$
    $\forall o : MiddleEl[o] = \texttt{Next}^*[o] \setminus \texttt{Next}^*[nextInst[o]]\,;$
    $ElLast = Ie.\texttt{Next}^*\,;$
Pre-conditions:
    $Ie.\texttt{next} \neq \textbf{null}\,;$
    $Ib \in Inst\,;$
    $Ie \in Inst\,;$
    $\textbf{null} \notin nextInst(Inst \setminus \{Ie\})\,;$

    $Inst = nextInst^*[Ib] \cup \{Ib\}\,;$
    $\forall o \in Inst \setminus \{Ie\} : nextInst^*[o] = nextInst^*[nextInst[o]] \cup nextInst[o]\,;$
    $nextInst^*[Ie] = \varnothing\,;$

    $Inst \subseteq Ib.Elements;$
    $\forall o \in Inst \setminus \{Ie\} : nextInst[o] \in o.\texttt{Next}^*\,;$
Transformations:
    $\langle ElFirst, Elements, ElFirst\rangle\,;\quad \langle ElLast, Elements, ElLast\rangle\,;$
    $\langle (Inst \setminus \{Ie\})\langle o'\rangle.MiddleEl, Elements, MiddleEl[o']\rangle\,;$
    $\langle ElFirst, \texttt{Next}^*, \textbf{v}.\texttt{Next}^* \setminus Ib.\texttt{Next}^*\rangle\,;$
    $\langle (Inst \setminus \{Ie\})\langle o'\rangle.MiddleEl, \texttt{Next}^*, \textbf{v}.\texttt{Next}^* \setminus nextInst[o'].\texttt{Next}^*\rangle\,;$
    $\langle ElLast, \texttt{Prev}^*, \textbf{v}.\texttt{Prev}^* \setminus Ie.\texttt{Prev}^* \cup \{Ie\}\rangle\,;$
    $\langle (Inst \setminus \{Ie\})\langle o'\rangle.MiddleEl, \texttt{Prev}^*, \textbf{v}.\texttt{Prev}^* \setminus o'.\texttt{Prev}^* \cup \{o'\}\rangle\,;$
    $\langle ElFirst, \texttt{last}, Ib\rangle\,;$
    $\langle (Inst \setminus \{Ie\})\langle o'\rangle.MiddleEl, \texttt{last}, nextInst[o']\rangle\,;$
    $\langle ElLast, \texttt{first}, Ie.\texttt{next}\rangle\,;$
    $\langle (Inst \setminus \{Ie\})\langle o'\rangle.MiddleEl, \texttt{first}, o'.\texttt{next}\rangle\,;$
    $\langle Inst, \texttt{next}, \textbf{null}\rangle$
    $\langle \textbf{v} \neq \textbf{null} \wedge \textbf{v}.\texttt{prev} \in Inst, \texttt{prev}, \textbf{null}\rangle$
Input instances: $\{Ib.\texttt{instID}\}$
Frame:
    $\langle ElLast, \texttt{last}\rangle\,;$
    $\langle ElFirst, \texttt{first}\rangle\,;$
    $\langle ElLast, \texttt{Next}^*\rangle\,;$
    $\langle ElFirst, \texttt{Prev}^*\rangle\,;$
Measure: $Inst$
{
    if$(Ib \neq Ie)$ then
      removeSetSequenceRelation$(Ib', Ie, Inst', nextInst)$
        measure $Inst' \overset{Ib}{\subset} Inst\,;$
    removeSequenceRelation$(Ib)\,;$
}
```

Figure A.2: `removeSetSequenceRelation` stereotype operation.

next element after Ie. Ib and Ie are the elements of $Inst$. The image of all elements of $Inst$ excluding Ie under $nextInst$ is not null.

The next subsection establishes the properties of $nextInst^*$. $nextInst^*$ of the first cut-point plus the first cut-point is equal to $Inst$. For each cut-point but the last one, $nextInst^*$ of the cut-point is equal to the $nextInst^*$ of the next cut-point plus the next cut-point. $nextInst^*$ of the last cut-point is the empty set.

The last subsection consists of two pre-conditions. The first one states that all cut-points are also elements of the input sequence instance. The next one states that for each o from $Inst \setminus \{Ie\}$, $nextInst[o]$ is reachable from o by `next`. This is part of the characteristic property of $nextInst$.

Let us now look at the transformation section of `removeSetSequenceRelation`. We group the transformation rules by the stereotype item which they update.

- *Elements*: There are three transformation rules which affect the values of *Elements*. They state that: the elements of the first output sequence are $ElFirst$, the elements of the middle output sequences are equal to $MiddelEl$, and the elements of the last output sequence are $ElLast$.
- `Next`*: For the elements of the first and the middle output sequences we decrease `Next`* by removing all elements after the next cut-point. For the first output sequence the next cut-point is Ib. For the middle output sequences we use $nextInst$ to identify the next cut-point.
- `Prev`*: For elements of the last and the middle output sequences we decrease `Prev`* by removing all elements before the cut-point and the cut-point. For the last output sequence the cut-point is Ie. For the middle output sequences the cut-point is o'.
- `last`: We set the last elements of the first and the middle output sequences to the next cut points.
- `first`: We set the first elements of the last and the middle output sequences to the successors of the cut-points.
- `next`: We set `next` of all cut-points to null.
- `prev`: We set `prev` of all successors of cut-points to null. Since the value of **v**.`prev` is undefined if $\mathbf{v} = \mathbf{null}$, we explicitly add the constraint $\mathbf{v} \neq \mathbf{null}$ to the transformation rule.

The input instance section states that there is only one affected sequence whose id is extracted from Ib.

The frame rules section is pretty straightforward. Values `first` and `Next`* are preserved for the first output sequence. Values `last` and `Prev`*

are preserved for the last output sequence. Since the values of *Elements* are updated for all relevant objects there is no frame rule for *Elements*. The frame rules for `next` and `prev` are inferred from the corresponding transformation rules.

The measure section states that the input parameter $Inst$ is used as a measure variable.

The last section is the method body. If $Inst$ is not a singleton set, which is the case if and only if $Ib \neq Ie$, then we make a recursive call which splits the tail of the input sequence. The last statement of the body is a call of `removeSequenceRelation` which separates the first output sequence from the rest of the input sequence.

Appendix B

Universal transformations for the tree stereotype

In this appendix we construct the universal transformations for the tree stereotype. As we have mentioned in **Section 6.5** it is enough to construct the universal set of relation addition and removal to get a universal transformation.

B.1 Level 1

B.1.1 Operation `addSetTreeRelation1`

- `Operation description:` The operation is the extension of `addTreeRelation` in case we add an unbounded number of sub-trees to an element of a tree. The operation is defined in Figure B.1. The result of the operation is depictured in Figure 4.25.
- `Input parameters:`
  - l: an element of the tree to which sub-tress are added.
  - P: set of instance identifiers of added sub-trees.
- `Local variables:`
  - p: an arbitrary element of P. If P is the empty set, the value of p is undefined.
  - P': the value of P for the recursive call.
  - PEL: the union of the elements of added sub-tress.
  - $newEl$: elements of the resulting tree. It is received by addition of the elements of the added sub-trees to an old element of the tree.

```
addSetTreeRelation1⟨Tree⟩($l$ : $ref!$, $P$ : $Reg!$)
Local variables:
    $p \in P$ ;
    $P' = P \setminus \{p\}$ ;
    $PEL = \mathbf{InstEl}(P)$ ;
    $newEl = l.Elements \cup PEl$
    $lA = \{l\} \cup l.$Anc
    $\forall o, o' : f'_{\texttt{Desc}}[o, o'] =$ if$(o = l)$ then $o'$.root else $o.f_{\texttt{Desc}}[l]$ ;
Pre-conditions:
    $\forall p \in P : p.$instID $= p$
    $l.$instID $\notin P$
Transformations:
    $\langle l,$ Child$, l.$Child $\cup P\rangle$ ;
    $\langle P,$ parent$, l\rangle$ ;
    $\langle PEl,$ root$, l.$root$\rangle$ ;
    $\langle newEl, Elements, newEl\rangle$ ;
    $\langle lA,$ Desc$, \mathbf{v}.$Desc $\cup PEl\rangle$ ;
    $\langle PEl,$ Anc$, \mathbf{v}.$Anc $\cup lA\rangle$ ;
    $\langle lA \times PEL, f_{\texttt{Desc}}, f'_{\texttt{Desc}}[v, v']\rangle$ ;
Input instances: $P \cup \{l.$instID$\}$ ;
Frame: $\varnothing$ ;
Measure: $P$
{
    if$(P \neq \varnothing)$ then{
        if$(P' \neq \varnothing)$ then addSetTreeRelation1$(l, P')$ measure $P' \overset{p}{\subset} P$ ;
        addTreeRelation$(p, l)$ ;
    }
}
```

Figure B.1: addSetTreeRelation1 stereotype operation.

  - lA: elements of the tree whose descendants are updated by the operation execution.
  - $f'_{\texttt{Desc}}$: an updated version of the $f_{\texttt{Desc}}$ function. We introduce the function to merge two transformation rules into one. For each o and o' such that o' is a descendent of o, $f'_{\texttt{Desc}}[o, o']$ returns a child of o through which o reaches o'. If the first parameter is l then such an element is the root of a corresponding added sub-tree. Since in all other cases the path from an element of the tree to an element of an added sub-tree goes through l the result is equal to $o.f_{\texttt{Desc}}[l]$. The reasoning is pretty much the same as for the `addTreeRelation` operation.

- `Pre-conditions:` We refer to a pre-condition by its line number in the operation definition.

  10. P contains only instances identifiers.
  11. l is disjoint form trees from P.

- `Transformations:`

  - `Child`: add P to the children of l.
  - `parent`: set parents of roots of sub-trees from P to l. Here we exploit the fact that the instance identifier of a tree is equal to the root of the tree.
  - `root`: set root of all elements of added sub-trees to l.
  - *Elements*: set values of *Elements* of all elements of the resulting tree into $newEl$.
  - `Desc`: add the union of elements of all added sub-trees (PEL) to `Desc` of l and its ancestors.
  - `Anc`: add lA to `Anc` of all elements of the added sub-tress.
  - $f_{\texttt{Desc}}$: We have to update $f_{\texttt{Desc}}$ for a pair of objects $\langle o, o' \rangle$ if and only if we add o' to the descendants of o. If we look at the transformation rule we can see that all such pairs of objects belong to $lA \times PEL$. We set a new value of $f_{\texttt{Desc}}$ into $f'_{\texttt{Desc}}$.

- `Input instances:` The input instances are P and the one which contains l.

- `Frame:` Since we do not use quantifiers in the transformation rules, all frame rules are inferred.

- `Measure:` We use P as a measure.

- `Body:` If P is the empty set then nothing has to be done. Otherwise, if P' is not the empty set we make a recursive call and use `addTreeRelation` to add the $\langle p, l\rangle$ relation. Since P contains p and P' does not, we know that the recursion terminates .

B.1.2 Operation `removeSetTreeRelation1`

- `Operation description:` The operation is dual to `saddSetTreeRelation1`. It removes a set of sub-trees which have the same parent. We also can think of the operation as the generalization of `removeTreeRelation`. The operation is defined in Figure B.2.
- `Input parameters:`
  - P: set of roots of removed sub-trees.
- `Local variables:`
  - p: an arbitrary element of P. If P is the empty set, the value of p is undefined.
  - P': the value of P for the recursive call.
  - l: an element of the tree from which the sub-trees are removed. If P is the empty set, the value of l is undefined.
  - $ElSub$: contains the elements of the subtrees.
  - f_{PEl}: is a witness function which is used to define the union of the elements of the removed sub-trees.
  - PEL: the union of the elements of the added sub-tress. Here we use the witness function f_{PEl}.
  - $newEl$: the elements of the tree after removal of sub-trees.
  - lA: the elements of the tree whose descendants are updated by the operation execution.
- `Pre-conditions:` The pre-condition states that if P is a non empty set then all elements of P are children of l. The pre-condition is equivalent to the following: all elements of P have the same parent.
- `Transformations:`
  - `Child`: remove P from the children of l.
  - `parent`: Since the elements of P are parents of the deleted subtrees, we set their parents to **null**.
  - `root`: For each $o' \in P$, o' is the new root of the corresponding sub-tree. We set `root` for all $ElSub[o']$, which contains elements of the corresponding sub-tree, to o'.

```
removeSetTreeRelation1⟨Tree⟩($P$ : $\mathit{Reg!}$)
Local variables:
    $p \in P$ ;
    $P' = P \setminus \{p\}$ ;
    $l = p.\texttt{parent}$ ;
    $\forall o : \mathit{ElSub}[o] = o.\texttt{Desc} \cup \{o\}$ ;
    $\forall o : f_{PEl}[o] = l.f_{\texttt{Desc}}[o]$ ;
    $PEl = \bigcup_{p \in P}^{f_{PEl}} \mathit{ElSub}[p]$ ;
    $\mathit{newEl} = l.\mathit{Elements} \setminus PEl$ ;
    $lA = \{l\} \cup l.\texttt{Anc}$ ;
Pre-conditions:
    $P \neq \varnothing \Rightarrow P \subseteq l.\texttt{Child}$
Transformations:
    $\langle l, \texttt{Child}, l.\texttt{Child} \setminus P \rangle$ ;
    $\langle P, \texttt{parent}, \mathbf{null} \rangle$ ;
    $\langle P\langle o' \rangle.\mathit{ElSub}, \texttt{root}, o' \rangle$ ;
    $\langle \mathit{newEl}, \mathit{Elements}, \mathit{newEl} \rangle$ ;
    $\langle P\langle o' \rangle.\mathit{ElSub}, \mathit{Elements}, \mathit{ElSub}[o'] \rangle$ ;
    $\langle lA, \texttt{Desc}, \mathbf{v}.\texttt{Desc} \setminus PEl \rangle$ ;
    $\langle PEl, \texttt{Anc}, \mathbf{v}.\texttt{Anc} \setminus lA \rangle$ ;
Input instances: $\{l.\texttt{instID}\}$ ;
Frame: $\varnothing$ ;
Measure: $P$
{
    if($P \neq \varnothing$) then{
        if($P' \neq \varnothing$) then removeSetTreeRelation1($l, P'$)
            measure $P' \overset{p}{\subset} P$ ;
        removeTreeRelation($p$) ;
    }
}
```

Figure B.2: `removeSetTreeRelation1` stereotype operation.

  - *Elements*: sets values of *Elements* of all elements of the tree to $newEl$. It also sets the values of *Elements* of the elements of the sub-tree with the root $o' \in P$ to $ElSub[o']$.
  - `Desc`: removes the union of the elements of all added sub-trees (PEL) from `Desc` of l and its ancestors.
  - `Anc`: removes lA from `Anc` of all elements of the added sub-tress.

- `Input instances:` One which contains l.
- `Frame:` The only stereotype item for which we use quantification in the transformation rules is *Elements*. But since its values are updated for all relevant elements we do not need a frame rule for it. For all other stereotype items, the frame rules are inferred.
- `Measure:` We use P as a measure.
- `Body:` If P is the empty set then nothing has to be done. Otherwise, if P' is not the empty set we make the recursive call and use `removeTreeRelation` to remove the $\langle p, l\rangle$ relation. Since P contains p and P' does not, we know that the recursion terminates.

B.2 Level 2

B.2.1 Operation `addSetTreeRelation2`

- `Operation description:` The operation is an extension of `addSetTreeRelation1` in case we add sub-trees to an unbounded number of elements of a tree. The operation is defined in Figure B.3. The result of the operation execution is depicted in Figure 4.26. To make the operation feasible we require a specification developer to provide an auxiliary tree relation. There are two kinds of elements of the auxiliary tree relation. The first one is the elements of the tree to which the sub-trees are added. The second one is the joint points of the auxiliary tree relation. o is a joint point of the auxiliary tree relation if and only if there are such elements of the auxiliary tree relation o_1 and o_2 that o_1 and o_2 are descendants of o, o_1 and o_2 belongs to different sub-trees of o, and there are no other elements of the auxiliary tree relation between o and o_1, and between o and o_2. In Figure 4.26 we denote the elements of the auxiliary tree relation with black circles. The solid lines between the black circles denote the auxiliary tree relation.
- `Input parameters:`
  - r_0: is the root of the auxiliary tree relation.
  - R: contains the elements of the the auxiliary tree relation excluding r_0.

```
addSetTreeRelation2⟨Tree⟩
        $(r_0 : ref!, R : Reg!, P : Reg!, r2P : ref \to TReg, p2r : ref \to ref,$
        $r2R : ref \to Reg, c2r : ref \to ref)$
Local variables:
        $R_0 = R \cup \{r_0\}$ ;
        $R_0^+ = R_0 \cup \{\mathbf{null}\}$ ;
        $\forall r : r2R^+[r] = \texttt{if}(r = \mathbf{null})\ \texttt{then}\ \{r_0\}\ \texttt{else}\ r2R[r]$ ;
        $\forall r : rA_0[r] = \texttt{if}(r = \mathbf{null})\ \texttt{then}\ \varnothing\ \texttt{else}\ r.\texttt{Anc} \cup \{r\}$ ;
        $\forall r, r' : rA[r, r'] = rA_0[r] \setminus rA_0[r']$ ;
        $\forall r : r2R^*[r] = R \cap r.\texttt{Desc}$ ;
        $\forall r : r2R^\top[r] = r2R[r] \cap r.\texttt{Child}$ ;
        $\forall r : r2R^\perp[r] = r2R[r] \setminus r2R^\top[r]$ ;
        $\forall r : r2C[r] \overset{c2r}{=} r.f_{\texttt{Desc}}(r2R^\perp[r])$ ;
        $\forall r : r2C_{All}[r] = r2R^\top[r] \cup r2C[r]$ ;
        $\forall r, r' : f_{r2r^*}[r][r'] = \texttt{if}(r' \in r2R^\top[r])\ \texttt{then}\ r'\ \texttt{else}\ c2r[r.f_{\texttt{Desc}}[r']]$ ;
        $\forall p : f_{p2r}[p] = f_{r2r^*}[r_0][p2r[p]]$ ;
        $\forall r, p : p \in r2P^*[r] \Leftrightarrow p \in P \wedge p2r[p] \in r2R^*[r]$ ;
        $\forall r : PEl[r] = \mathbf{InstEl}(r2P^*[r])$ ;
        $newEl = r_0.Elements \cup PEL[r_0]$ ;
        $\forall o : o2r[o] = p2r[o.\texttt{instID}]$ ;
        $\forall o, o' : f'_{\texttt{Desc}}[o, o'] = \texttt{if}(o = o2r[o'])\ \texttt{then}\ o'.\texttt{root}\ \texttt{else}\ o.f_{\texttt{Desc}}[o2r[o']]$ ;
Pre-conditions:
        $r_0 \notin R$ ;
        $R \subseteq r_0.\texttt{Desc}$ ;
        $\forall r \in R_0 : r2R[r] \subseteq r.\texttt{Desc}$ ;
        $\forall r \in R_0 : r2r^*[r] = (\bigcup^{f_{r2r^*}[r]}_{r' \in r2R[r]} r2r^*[r']) \cup r2r[r]$ ;
        $\forall p \in P : p.\texttt{instID} = p$ ;
        $r_0.\texttt{instID} \notin P$ ;
        $\forall p \in P : p2r[p] \in R_0$ ;
        $\forall r \in R_0 : r2P[r] \subseteq P$ ;
        $\forall p \in P : p \in r2P[p2r[p]]$ ;
        $\forall r \in R_0, p \in r2P[r] : p2r[p] = r$ ;
Transformations:
        $\langle R_0, \texttt{Child}, \mathbf{v}.\texttt{Child} \cup r2P[\mathbf{v}]\rangle$ ;
        $\langle P, \texttt{parent}, p2r[\mathbf{v}]\rangle$ ;
        $\langle PEL[r_0], \texttt{root}, r_0.\texttt{root}\rangle$ ;
        $\langle newEl, Elements, newEl\rangle$ ;
        $\langle \exists r, r' : r' \in R_0^+ \wedge r \in r2R^+[r'] \wedge \mathbf{v} \in rA[r, r'], \texttt{Desc}, \mathbf{v}.\texttt{Desc} \cup PEL[r]\rangle$ ;
        $\langle P, \texttt{Anc}, rA_0[p2r[\mathbf{v}]]\rangle$ ;
        $\langle P\langle p\rangle.\texttt{Desc}, \texttt{Anc}, \mathbf{v}.\texttt{Anc} \cup rA_0[p2r[p]]\rangle\rangle$ ;
        $\langle \exists r, r' : r' \in R_0^+ \wedge r \in r2R^+[r'] \wedge \mathbf{v} \in rA[r, r'] \wedge \mathbf{v'} \in PEL[r],$
                $f_{\texttt{Desc}}, f'_{\texttt{Desc}}[\mathbf{v}, \mathbf{v'}]\rangle$ ;
Input instances: $\{r_0.\texttt{instID}\} \cup P$
Frame:
        $\langle \mathbf{v}.\texttt{Desc} \cup \{\mathbf{v}\} \sharp R_0, \texttt{Desc}\rangle$ ;
        $\langle r_0.Elements, \texttt{Anc}\rangle$ ;
        $\langle \mathbf{v}.\texttt{Desc} \cup \{\mathbf{v}\} \sharp R_0, f_{\texttt{Desc}}\rangle$ ;
        $\langle \exists r, r' : r' \in R_0^+ \wedge r \in r2R^+[r'] \wedge \mathbf{v} \in rA[r, r'] \wedge \mathbf{v'} \notin PEL[r], f_{\texttt{Desc}}\rangle$ ;
Measure: $R$
{
        if($R \neq \varnothing$) then {
                removeSetTreeRelation1($r2C_{All}[r_0]$);
                $\mathop{\|}^{f_{p2r}}_{r \in r2R[r_0]}$ addSetTreeRelation2($r, r2R^*[r], r2P^*[r],$
                        $r2P, p2r, r2R, c2r$) measure $r2R^*[r] \overset{r_0}{\subset} R$ ;
                addSetTreeRelation1($r_0, r2C_{All}[r_0]$);
        }
        addSetTreeRelation1($r_0, r2P[r_0]$);
}
```

Figure B.3: addSetTreeRelation2 stereotype operation.

  - P: the set of the instance identifiers of the added sub-trees.
  - $r2P$: a map from an element of the auxiliary tree relation to a set of instance identifiers of the added sub-trees to the element.
  - $p2r$: an inversion of $r2P$.
  - $r2R$: a map from an element of the auxiliary tree relation to the set of its children in the auxiliary tree relation.
  - $c2r$: an inversion of $r.f_{\texttt{Desc}}$ for each r from R_0. We use it as a witness function to define an auxiliary map $r2C$.

- `Local variables:`
  - R_0: all elements of the auxiliary tree relation including r_0.
  - R_0^+: R_0 plus **null**. In some cases, which we explain below, we use **null** as a special marker for a parent of r_0 in the auxiliary tree relation. It helps to keep the definitions brief.
  - $r2R^+$: an extension of $r2R$ on **null**. We define $r2R[\textbf{null}]$ as $\{r_0\}$. The only child of **null** in the auxiliary tree relation is r_0.
  - $rA_0[r]$: for an element of R_0 the map returns the set which contains the element and its ancestors. For **null** it returns the empty set.
  - rA: contains all tree elements between two elements of the auxiliary tree relation. For r_0 it returns all elements from r_0 until the root of the tree.
  - $r2R^*$: the transitive closure of $r2R$
  - $r2R^\top$: the children of an element of the auxiliary tree relation which are also children of the element in the tree relation.
  - $r2R^\perp$: the children of an element of the auxiliary tree relation which are not children of the element in the tree relation.
  - $r2C$: for an element r of the auxiliary tree relation, $r2C[r]$ contains a subset of $r.\texttt{Child}$ such that: for each $c \in r2C[r]$ $c.\texttt{Desc}$ contains exactly one element of $r2R^\perp[r]$ and for each element of $r2R^\perp[r]$ there is exactly one $c \in r2C[r]$ such that $c.\texttt{Desc}$ contains the element. We define $r2C[r]$ as an image of $r2R^\perp[r]$ under $r.f_{\texttt{Desc}}$. As we have mentioned above we use $c2r$ as a witness function to define $r2C$.
  - $r2C_{All}$: a map from an element r of the auxiliary tree relation into a sub-set of $r.\texttt{Child}$ such each element of the subset either a child or an ancestor of a child of r in the auxiliary tree relation. We use $r2C_{All}$ to split the tree in the sub-trees in a way that the effects of the recursive calls in the body are disjoint and can be executed in parallel.

- f_{r2r^*}: is a witness function which we use to check that $r2R^*$ has a proper tree structure.
- f_{p2r}: is a witness function which is used by the parallel composition. It maps an instance identifier p of an added sub-tree to the child r of r_0 in the auxiliary tree relation which is the ancestor of the r' to which the sub-tree with identifer p is added.
- $r2P^*$: maps an element of the auxiliary tree relations to the set of instance identifiers of the sub-tress which are added to the element.
- PEl: union of elements of sub-tress which are added to a specific element of the auxiliary tree relation.
- $newEl$: elements of the resulting tree.
- $o2r$: a map from an element of a sub-tree to an element of the tree to which it is added.
- $f'_{\texttt{Desc}}[o, o']$: an updated version of $f_{\texttt{Desc}}$ function. The definition is similar to the corresponding definition from `addSetTreeRelation1`.

- `Pre-conditions:`

23. r_0 doesn't belong to R.
24. each element of R is a descendant of r_0.
25. for each element r of the auxiliary tree relation, the children of r in the auxiliary tree relation are descendants of r in the tree.
26. for each element r of the auxiliary tree relation, the descendants of r in the auxiliary tree relation are equal to the union of the children of r in the auxiliary tree relation and the union of the descendant of the children of r in the auxiliary tree relation. The property is an inductive hypotheses. It could be inferred from the other properties. Nevertheless, such an inference requires the construction of f_{r2r^*} which could be problematic for an automatic theorem prover. We use the property in the body of the operation to facilitate verification of inductive calls.
27. P is a set of instance identifiers.
28. the instance identifier of r_0 doesn't belong to P.
29. the image of P under $p2r$ is a subset of R_0. The property checks that $p2r$ has a proper range.
30. the image of R_0 under $r2P$ is a subset of P. The property checks that $r2P$ has a proper range.
31. $r2P$ is the inverse of $p2r$.

32. $p2r$ is the inverse of $r2P$.

- `Transformations:`
  - `Child`: to each element $\mathbf{v}$ of the auxiliary tree relation, add to the children of $\mathbf{v}$ the roots of all added sub-trees, which are equal to $r2P[\mathbf{v}]$.
  - `parent`: set the parents of roots of the added sub-trees to the elements of the tree to which they are added.
  - `root`: set the root of all elements of all added sub-trees to the root of r_0.
  - *Elements*: set *Elements* of the final tree to $newEl$
  - `Desc`: for each element r' of the auxiliary tree relation and each child r of r' in the auxiliary tree relation, we add to the descendants of all elements between r and r', which is equal to $rA[r, r']$, the union of elements of all sub-trees added to r or its descendants. A special case when $r' = \mathbf{null}$ describes updates of descendants of all elements from r_0 to r_0`.root`.
  - `Anc`: for each added sub-tree with instance identifier p we add $rA_0[p2r[p]]$ to `Anc` of all elements of the sub-tree. $rA_0[p2r[p]]$ contains an element of the tree to which the p sub-tree is added and all its ancestors. There are two transformation rules which affect `Anc`: one updates the values of the roots of the sub-trees, and the other one affects all other elements of the sub-trees.
  - $f_{\mathtt{Desc}}$: $f_{\mathtt{Desc}}$ is updated for all pairs $\langle \mathbf{v}, \mathbf{v'} \rangle$ where $\mathbf{v'}$ is added to the descendants of $\mathbf{v}$. That is why the rule is very similar to the one which updates the values of `Desc`. The only extra part of the rule states that $\mathbf{v'} \in PEL[r]$. Here $PEL[r]$ are the elements which are added to $\mathbf{v}$`.Desc`. The new value of $f_{\mathtt{Desc}}$ is $f'_{\mathtt{Desc}}$.

- `Input instances:` The input instances are P and the one which contains r_0.

- `Frame:`
  - `Desc`: `Desc` of an object $\mathbf{v}$ is changed if and only if $\mathbf{v}$ belongs to R_0 or its ancestors. From this observation we infer the following frame rule: `Desc` of $\mathbf{v}$ is preserved if and only if $\mathbf{v}$ and its descendants are disjoint from R_0.
  - `Anc`: the value of `Anc` is preserved for all elements of the tree which contains r_0.
  - $f_{\mathtt{Desc}}$: there are two frame rules for $f_{\mathtt{Desc}}$. There first one states that if the value `Desc` of an object $\mathbf{v}$ is preserved then the value

of $f_{\texttt{Desc}}$ is also preserved for $\mathbf{v}$. The second one states that if $\mathbf{v}$ is located between the elements of the auxiliary tree relation r' and r but **v'** does not belong to $PEL[r]$ then the value of $\mathbf{v}.f_{\texttt{Desc}}[\mathbf{v'}]$ is preserved.

  – *Elements*, `Child`, `parent`, and `root`: there are no frame rules for these elements since they are automatically inferred from the transformation rules.

- `Measure:` as a measure we use input parameter R.

- `Body:` if $R = \varnothing$ then we skip the recursive part of the operation. Otherwise we remove all relations which originate from $r2C_{All}[r_0]$ by calling `removeSetTreeRelation1`. By this we split the tree in a set of disjoint trees. Then we make a recursive call for each of the split trees. Since the trees are disjoint we can execute all recursive calls in parallel. Each tree in the parallel composition is identified by an r from $r2R[r_0]$. The values of input parameters R and P in the recursive calls are equal to $r2R^*[r]$ and $r2P^*[r]$, respectively. Since the rest of the parameters are maps and the recursive calls rely on their properties on smaller domains we can reuse them without changes. After the recursive calls we merge the split trees by calling `addSetTreeRelation1`. The last statement of the body uses `addSetTreeRelation1` to add sub-trees to r_0.

B.2.2 Operation `removeSetTreeRelation2`

- `Operation description:` The operation is dual to `addSetTreeRelation2`. It removes a set of sub-trees which possibly have different parents. The only limitation is that we can't remove a sub-tree from another removed sub-tree. We can also think about the operation as a generalization of `removeSetTreeRelation1`. The operation is defined on Figure B.4.

- `Input parameters:` Most of the input parameters have the same meaning as for `addSetTreeRelation2`. The only extra parameter is f_{PEL}. f_{PEL} for each element of a removed sub-tree returns the root of the sub-tree. We use f_{PEL} as a witness function to compute the union of the elements of the removed sub-trees. Another difference is that P are elements of the same input tree instance. $r2P$ and $p2r$ are not among the input parameters because information about them can be extracted from the `Child` and `parent` relations.

- `Local variables:` Most of the local variables have the same meaning and definition as for `addSetTreeRelation2`. Let us consider those which differ:

```
removeSetTreeRelation2⟨Tree⟩($r_0 : ref!, R : Reg!, P : Reg!,$
        $r2R : ref \to Reg, c2r : ref \to ref, f_{PEL} : ref \to ref$)
Local variables:
        $R_0 = R \cup \{r_0\}$ ;
        $\forall r : r2P[r] = P \cap r.\texttt{Child}$ ;
        $R_0^+ = R_0 \cup \{\textbf{null}\}$ ;
        $\forall r : r2R^+[r] =$ if$(r = \textbf{null})$ then $\{r_0\}$ else $r2R[r]$ ;
        $\forall r : rA_0[r] =$ if$(r = \textbf{null})$ then $\varnothing$ else $r.\texttt{Anc} \cup \{r\}$ ;
        $\forall r, r' : rA[r, r'] = rA_0[r] \setminus rA_0[r']$ ;
        $\forall r : r2R^*[r] = R \cap r.\texttt{Desc}$ ;
        $\forall r : r2R^\top[r] = r2R[r] \cap r.\texttt{Child}$ ;
        $\forall r : r2R^\perp[r] = r2R[r] \setminus r2R^\top[r]$ ;
        $\forall r : r2C[r] \overset{c2r}{=} r.f_{\texttt{Desc}}(r2R^\perp[r])$ ;
        $\forall r : r2C_{All}[r] = r2R^\top[r] \cup r2C[r]$ ;
        $\forall r, r' : f_{r2r^*}[r][r'] =$ if$(r' \in r2R^\top[r])$ then $r'$ else $c2r[r.f_{\texttt{Desc}}[r']]$ ;
        $\forall p : f_{p2r}[p] = f_{r2r^*}[r_0][p.\texttt{parent}]$ ;
        $\forall r : r2P^*[r] = P \cap r.\texttt{Desc}$ ;
        $\forall p : ElSub[p] = p.\texttt{Desc} \cup \{p\}$ ;
        $\forall r : PEl[r] = \bigcup_{p \in r2P^*[r]}^{f_{PEL}} ElSub[p]$ ;
        $newEl = r_0.Elements \setminus PEL[r_0]$ ;
Pre-conditions:
        $r_0 \notin R$ ;
        $R \subseteq r_0.\texttt{Desc}$ ;
        $\forall r \in R_0 : r2R[r] \subseteq r.\texttt{Desc}$ ;
        $\forall r \in R_0 : r2r^*[r] = \left( \bigcup_{r' \in r2R[r]}^{f_{r2r^*[r]}} r2r^*[r'] \right) \cup r2r[r]$ ;
        $\forall r \in R_0 : P \sharp r2C_{All}[r]$ ;
        $\forall p \in P : p.\texttt{parent} \in R_0$ ;
Transformations:
        $\langle R_0, \texttt{Child}, \mathbf{v}.\texttt{Child} \setminus r2P[\mathbf{v}] \rangle$ ;
        $\langle P, \texttt{parent}, \textbf{null} \rangle$ ;
        $\langle P\langle o' \rangle.ElSub, \texttt{root}, o' \rangle$ ;
        $\langle newEl, Elements, newEl \rangle$ ;
        $\langle P\langle o' \rangle.ElSub, Elements, ElSub[o'] \rangle$ ;
        $\langle \exists r, r' : r' \in R_0^+ \wedge r \in r2R^+[r'] \wedge \mathbf{v} \in rA[r, r'], \texttt{Desc}, \mathbf{v}.\texttt{Desc} \setminus PEL[r] \rangle$ ;
        $\langle P, \texttt{Anc}, \varnothing \rangle$ ;
        $\langle P\langle p \rangle.\texttt{Desc}, \texttt{Anc}, \mathbf{v}.\texttt{Anc} \setminus p.\texttt{Anc} \rangle$ ;
Input instances: $\{r_0.\texttt{instID}\}$
Frame:
        $\langle \mathbf{v}.\texttt{Desc} \cup \{\mathbf{v}\} \sharp R_0, \texttt{Desc} \rangle$ ;
        $\langle newEl, \texttt{Anc} \rangle$ ;
Measure: $R$
{
        if$(R \neq \varnothing)$ then {
                removeSetTreeRelation1$(r2C_{All}[r_0])$ ;
                $\Vert_{r \in r2R[r_0]}^{f_{p2r}}$ removeSetTreeRelation2$(r, r2R^*[r], r2P^*[r],$
                        $r2P, r2R, c2r, f_{PEL})$ measure $r2R^*[r] \overset{r_0}{\subset} R$ ;
                addSetTreeRelation1$(r_0, r2C_{All}[r_0])$ ;
        }
        removeSetTreeRelation1$(r2P[r_0])$ ;
}
```

Figure B.4: `removeSetTreeRelation2` stereotype operation.

  - $r2P$: $r2P[r]$ contains the children of r which also belong to P.
  - $r2P^*$: $r2P^*[r]$ contains the descendants of r which also belong to P.
  - f_{p2r}: the definition of f_{p2r} is similar to the corresponding definition from `addSetTreeRelation2`. The only difference is that instead of $p2r$ we use `parent`.
  - $ElSub$: are elements of the corresponding sub-tree. We define them as an origin of a removed relation and its descendants.
  - PEL: is the union of all elements of removed sub-trees which are reachable from r. $r2P^*[r]$ contains all roots of all removed subtres reachable from r. Here we use f_{PEL} as a witness function.
  - $newEl$: elements of the resulting tree. We define it is the old elements of the tree minus the elements of the removed sub-tress.

- `Pre-conditions:` The first four pre-conditions are the same as for `addSetTreeRelation2`. Let us consider the pre-conditions which differ:

  26. Since $r2C_{All}[r]$ contains the elements through which the edges of the auxiliary tree relation passes, P has to be disjoint from $r2C_{All}[r]$
  27. each element of P has to be a child of an element of R_0. By this we guarantee that we remove only sub-trees of R_0.

- `Transformations:`

  - `Child`: for each $\mathbf{v}$ from R_0 remove $r2P[\mathbf{v}]$ from the children of $\mathbf{v}$.
  - `parent`: set the parents of the roots of the removed sub-trees to **null**.
  - `root`: for each $o' \in P$, o' is the new root of the corresponding sub-tree. We set `root` for all $ElSub[o']$, which contains elements of the corresponding sub-tree, to o'.
  - *Elements*: set values of *Elements* of all elements of the tree to $newEl$. Set values of *Elements* of the elements of a sub-tree with the root $o' \in P$ to $ElSub[o']$.
  - `Desc`: for each element r' of the auxiliary tree relation and each child r or r' in the auxiliary tree relation we remove the union of the elements of all sub-trees removed from r or its descendants from descendants of all elements between r and r', which is equal to $rA[r, r']$.
  - `Anc`: for each removed sub-tree with root p we remove p and its ancestors from `Anc` of all elements of the sub-tree. There are two

transformation rules which affect `Anc`: one updates the values of the roots of the sub-trees, and the other affects all other elements of the sub-trees.

- `Input instances:` one which contains r_0.
- `Frame:`
  - `Desc`: similarly to `addSetTreeRelation2`, `Desc` of **v** is preserved if and only if **v** and its descendants are disjoint from R_0.
  - `Anc`: the value of `Anc` is preserved for all elements of the output tree which are equal to $newEl$.
  - *Elements*: since the values of *Elements* are updated for all relevant elements we don't need a frame rule for it.
  - `Child`, `parent`, $f_{\texttt{Desc}}$, and `root`: there are no frame rules for these elements since they are automatically inferred from the transformation rules.
- `Measure:` as a measure we use the input parameter R.
- `Body:` if $R = \varnothing$ then we skip the recursive part of the operation. Otherwise we remove all relations which originate from $r2C_{All}[r_0]$ by calling `removeSetTreeRelation1`. Then we make a recursive call for each of the split trees. Since the trees are disjoint we can execute all recursive calls in parallel. Each tree in the parallel composition is identified by an r from $r2R[r_0]$. The values of the input parameters R and P in the recursive calls are equal to $r2R^*[r]$ and $r2P^*[r]$, respectively. Since the rest of the parameters are maps and the recursive calls rely on their properties on smaller domains, we can reuse them without changes. After the recursive calls we merge the split trees by calling `addSetTreeRelation1`. The last statement of the body uses `removeSetTreeRelation1` to remove sub-trees from r_0.

B.3 Level 3

B.3.1 Operation `addSetTreeRelation3`

- `Operation description:` `addSetTreeRelation3` generalizes `addSetTreeRelation2` to the addition of an arbitrary set of relations which merge the input trees into one output tree. As an auxiliary input information the operation requires a description of a tree-of-trees relation. The elements of the relation are the merged trees. Two trees are in the relation if and only if one of them is added as a sub-tree to the other one. In Figure 4.27 we depict the result of the operation in case the height of the tree-of-trees is two. The triangles of the

```
addSetTreeRelation3⟨Tree⟩($r_0 : ref \to ref!, R_{All} : Reg!, P : ref \to Reg!,$
        $r2P : ref \to TReg, p2r : ref \to ref, r2R : ref \to Reg, c2r : ref \to ref,$
        $T : Reg!, t_0 : ref, tD : ref \to Reg!, f_{tD} : ref^2 \to ref$)
Local variables:
        $\forall t : R_0[t] = R_{All} \cap t.Elements$;
        $R^+_{All} = R_{All} \cup \{\mathbf{null}\}$;
        $\forall r : r2R^+[r] =$ if($r = \mathbf{null}$) then $\{r_0\}$ else $r2R[r]$;
        $\forall t : R[t] = R_0[t] \setminus \{r_0[t]\}$;
        $\forall r : rA_0[r] =$ if($r = \mathbf{null}$) then $\varnothing$ else $r.\mathtt{Anc} \cup \{r\}$;
        $\forall t, r : r2R^*[r] = R[r.\mathtt{instID}] \cap r.\mathtt{Desc}$;
        $\forall r : r2R^\top[r] = r2R[r] \cap r.\mathtt{Child}$;
        $\forall r : r2R^\perp[r] = r2R[r] \setminus r2R^\top[r]$;
        $\forall r : r2C_{All}[r] = r2R^\top[r] \cup r2C[r]$;
        $\forall t, t' : f'_{tD}[t, t'] =$ if($t' \in P[t]$) then $t'$ else $f_{tD}[t, t']$;
        $\forall p : f_{p2t}[p] = f'_{tD}[t, p2r[p].\mathtt{instID}]$;
        $\forall r, p : p \in r2P^*[r] \Leftrightarrow p \in T \wedge p2r[f'_{tD}[r.\mathtt{instID}, p]] \in r2R^*[r]$;
        $\forall r : PEl[r] = \mathbf{InstEl}(r2P^*[r])$;
        $newEl = \mathbf{InstEl}(T)$;
        $\forall t, t' : t' \in tA[t] \Leftrightarrow t \in tD[t']$;
        $\forall o : f_{inst}[o] = o.\mathtt{instID}$;
        $\forall t : tAU[t] = \bigcup^{f_{inst}}_{t' \in tA[t]} p2r[f'_{tD}[t', t]].\mathtt{Anc}$;
        $\forall o, o' : o2p[o, o'] = f'_{tD}[o.\mathtt{instID}, o'.\mathtt{instID}]$;
        $\forall o, o' : o2r[o, o'] = p2r[o2p[o, o']]$;
        $\forall o, o' : f'_{\mathtt{Desc}}[o, o'] =$
                if($o = o2r[o, o']$) then $o2p[o, o']$ else $o.f_{\mathtt{Desc}}[o2r[o, o']]$;
Pre-conditions:
        $\forall t \in T :$ addSetTreeRelation2($r_0[t], R[t], P[t], r2P, p2r, r2R, c2r$);
        $\forall t \in T : r_0[t] \in R_0[t]$;
        $\forall t \in T : t.\mathtt{instID} = t$;
        $T = \{t_0\} \cup tD[t_0]$;
        $\forall t \in T : tD[t] \subseteq T$;
        $\forall t \in T, t' \in P[t] : P[t] \sharp tD[t']$;
        $\forall t \in T, t_1 \in P[t], t_2 \in P[t] : tD[t_1] \sharp tD[t_2]$;
        $\forall t \in T : tD[t] = \left( \bigcup^{f_{tD}[t,.]}_{t' \in P[t]} tD[t'] \right) \cup P[t]$;
Transformations:
        $\langle R_{All}, \mathtt{Child}, \mathbf{v}.\mathtt{Child} \cup r2P[\mathbf{v}] \rangle$;
        $\langle T \setminus \{\ t_0\}, \mathtt{parent}, p2r[\mathbf{v}] \rangle$;
        $\langle PEL[r_0[t_0]], \mathtt{root}, t_0.\mathtt{root} \rangle$;
        $\langle newEl, Elements, newEl \rangle$;
        $\langle \exists r, r' : r' \in R^+_{All} \wedge r \in r2R^+[r'] \wedge \mathbf{v} \in rA[r, r'], \mathtt{Desc}, \mathbf{v}.\mathtt{Desc} \cup PEL[r] \rangle$;
        $\langle T \setminus \{t_0\}, \mathtt{Anc}, tAU[t] \rangle$;
        $\langle (T \setminus \{t_0\}) \langle t \rangle.\mathtt{Desc}, \mathtt{Anc}, \mathbf{v}.\mathtt{Anc} \cup tAU[t] \rangle \rangle$;
        $\langle \exists r, r' : r' \in R^+_{All} \wedge r \in r2R^+[r'] \wedge \mathbf{v} \in rA[r, r'] \wedge$
                $\mathbf{v'} \in PEL[r], f_{\mathtt{Desc}}, f'_{\mathtt{Desc}}[\mathbf{v}, \mathbf{v'}] \rangle$;
Input instances: $T$
Frame:
        $\langle \mathbf{v}.\mathtt{Desc} \cup \{\mathbf{v}\} \sharp R_{All}, \mathtt{Desc} \rangle$;
        $\langle t_0.Elements, \mathtt{Anc} \rangle$;
        $\langle \mathbf{v}.\mathtt{Desc} \cup \{\mathbf{v}\} \sharp R_{All}, f_{\mathtt{Desc}} \rangle$;
        $\langle \exists r, r' : r' \in R^+_{All} \wedge r \in r2R^+[r'] \wedge \mathbf{v} \in rA[r, r'] \wedge \mathbf{v'} \notin PEL[r], f_{\mathtt{Desc}} \rangle$;

Measure: $T$
{
        if($T \setminus \{t_0\} \neq \varnothing$) then {
                $\mathop{\|}^{f_{p2t}}_{t \in P[t_0]}$ addSetTreeRelation3($r_0, R[t], P, r2P, p2r, r2R, c2r,$
                                $tD[t] \cup \{t\}, t, tD, f_{tD}$) measure $tD[t_0] \overset{t_0}{\subset} T$;
        }
        addSetTreeRelation2($r_0[t_0], R[t_0], P[t_0], r2P, p2r, r2R, c2r$);
}
```

Figure B.5: `addSetTreeRelation3` stereotype operation.

different sizes denote the trees from the different levels of the tree-of-trees relation. The biggest triangle denotes the root of the tree-of-trees relation. Trees from the second level of the tree-of-trees relation, which are denoted by the triangles of the middle size, are added as the sub-trees to the various elements of the root of the tree-of-trees relation. The smallest triangles denote the trees which belong to the third level of the tree-of-trees relation. Additionally to the tree-of-trees relation a client has to specify the auxiliary tree relations for each tree. Similarly to `addSetTreeRelation2` we denote the nodes of the auxiliary tree relations as black circles and the relations as solid lines. `addSetTreeRelation3` is defined in Figure B.5.

- `Input parameters:`
  - r_0: a map form a tree identifier to the root of the auxiliary tree relation of the tree.
  - R_{All}: contains elements of the auxiliary tree relations of all trees.
  - P: a map form a tree identifier to the set of trees added to the tree. From the other perspective the relation maps an element of the tree-of-trees relation into its child in the relation.
  - $r2P$, $p2r$, $r2R$, and $c2r$: have the same meaning as for the `addSetTreeRelation2` operation.
  - T: the set of instance identifiers of tress which participate in the operation.
  - t_0: the root of the tree-of-trees relation.
  - tD: a map of an element of the tree-of-trees relation to its descendant in the relation.
  - f_{tD}: is an analog of $f_{\texttt{Desc}}$ for the tree-of-trees relation. For each pair $\langle t, t' \rangle$ of elements of the tree-of-trees relation, where t' is a descendant but not a child of t in the tree-of-trees relation, $f_{tD}[t, t']$ returns an element t'' of the relation such that t'' is a child of t and an ancestor of t'. We use f_{tD} as a witness function in some definitions.
- `Local variables:`
  - R_0: a map form a tree identifier to the elements of the auxiliary tree relation of the tree.
  - R^+_{All}: R^+_{All} plus **null**. Similarly to `addSetTreeRelation2` we use **null** as a special marker for a parent of $r_0[t]$ in the auxiliary tree relation of the tree with instance identifier t.
  - $r2R^+$: the definition is the same as for `addSetTreeRelation2`.

- R: for each tree identifier t, $R[t]$ is the same as $R_0[t]$ but without $r_0[t]$.
- rA_0: the definition is the same as for `addSetTreeRelation2`.
- $r2R^*$: the transitive closure of $r2R$. We define it as the elements of the auxiliary tree relation which are also descendants of r.
- $r2R^\top$, $r2R^\perp$, and $r2C_{All}$: the definition is the same as for `addSetTreeRelation2`.
- f'_{tD}: is an extension of $f_{tD}[t, t']$ to the case where t' is a child of t' in the relation tree-of-trees. In this case the extended map returns t' because it is the only child of t on the path between t and t'.
- f_{p2t}: f_{p2t} is an analog of f_{p2r} from `addSetTreeRelation2`. f_{p2t} is a witness function which is used by the parallel composition. It maps an instance identifier p of an added sub-tree to the child t of t_0 in the tree-of-trees relation which is an ancestor of the t' to which the sub-tree with identifer p is added.
- $r2P^*$: maps an element of the auxiliary tree relations to a set of instance identifiers of sub-tress which are added to the element and its descendants. The definition is similar to the corresponding one in `addSetTreeRelation2`. The only difference is that to identify a tree through which p is connected to p, f'_{tD} is used.
- PEl: the definition is the same as for `addSetTreeRelation2`.
- $newEl$: the definition is the same as for `addSetTreeRelation2`.
- tA: a map of an element of the tree-of-trees relation to its ancestors in the relation. tA is defined as the inverse of tD.
- f_{inst}: is an auxiliary function which returns the instance identifier of the input parameter.
- $tAU[t]$: a map of an element of the tree-of-trees relation to the ancestors of the root of the tree in the output tree. We use f_{inst} as a witness function and composition of $p2r$ and f'_{tD} to identify an element of the auxiliary relation tree-of-trees with tree identifier t' which is also an ancestor of t.
- $o2p$: for each pair $\langle o, o' \rangle$, where o'.`instID` is a descendant of o.`instID` in the tree-of-trees relation, $o2p[o, o']$ returns an element t'' of the relation such that t'' is a child of o.`instID` and t'' an ancestor of o'.`instID` or $t'' = o$.`instID`.
- $o2r$: similar to $o2p$ but instead of a tree instance identifier t'' it returns an element of the auxiliary tree relation to which the tree with identifier t'' is added.

- $f'_{\texttt{Desc}}$: an updated version of the $f_{\texttt{Desc}}$ function. The definition is similar to the corresponding definition from `addSetTreeRelation2`. The only difference is that we use $o2p$ and $o2r$ to identify the corresponding elements of the tree-of-trees and auxiliary tree relations.

- `Pre-conditions:`

  27. for each tree which participates in the operation all pre-conditions from `addSetTreeRelation2` hold.
  28. for all tree identifiers t, $r_0[t]$ is an element of $R_0[t]$.
  29. T is a set of tree instances.
  30. an element of the tree-of-trees relation is either the root of the relation or a descendant of the root.
  31. for each element of the tree-of-trees relation, the descendants of the element are also elements of the tree-of-trees relation.
  32. for each element of the tree-of-trees relation, the children of the element are disjoint from the descendants of a child of the element.
  33. for each pair of siblings in the tree-of-trees relation their descendant are disjoint.
  34. for each element t of the tree-of-trees relation the descendants of t are equal to the union of children of t and the union of the descendant of the children of t. The property is an inductive hypotheses. It could be inferred from the other properties. Nevertheless, such an inference requires guessing that $f_{tD}[t,.]$ can be used as a witness function which could be problematic for an automatic theorem prover. We use the property in the body of the operation to facilitate the verification of inductive calls.

- `Transformations:` The transformation section is almost the same as for `addSetTreeRelation2`.

- `Input instances:` The input instances are T.

- `Frame:` The frame section is almost the same as for `addSetTreeRelation2`.

- `Measure:` as a measure we use input parameter T.

- `Body:` if $T \setminus \{t_0\} = \varnothing$ then we skip the recursive part of the operation. Otherwise we execute all recursive calls in parallel. Each tree in the parallel composition is identified by t from $P[t_0]$. The values of the input parameters R_{All} and T in the recursive calls are equal to $R[t]$

and $tD[t] \cup \{t\}$, respectively. Since rest of the parameters are maps and the recursive calls rely on their properties on smaller domains we can reuse them without changes. The last statement of the body uses `addSetTreeRelation2` to add sub-trees to t_0.

B.3.2 Operation `removeSetTreeRelation3`

- `Operation description:` The operation is dual to `addSetTreeRelation3`. It removes an arbitrary set of relations. We can also think of the operation as a generalization of `removeSetTreeRelation2`. Similarly to `addSetTreeRelation3` a client of the operation has to define the tree-of-trees relation and for each removed sub-tree an auxiliary tree relation. The operation is defined in Figure B.6.

- `Input parameters:`

  – p_0: the root of the tree-of-trees relation.
  – $p2r_0$: a map form the root of an output tree to the root of the auxiliary tree relation of the output tree.
  – R: contains the elements of the auxiliary tree relations of all removed trees.
  – P: the set of roots of the output trees.
  – $r2R$, $c2r$: have the same meaning as for `removeSetTreeRelation2`.
  – f_{PEL}: for each root of an output tree, $f_{PEL}[p, .]$ returns f_{PEL} for corresponding tree.
  – $p2P$: a map from the root of an output tree to the set of the roots of the trees removed from the tree. From the other perspective, $p2P[p]$ is equal to the children of p in the tree-of-trees relation.

- `Local variables:`

  – $r2P^*$: a map from an element of the auxiliary tree relation of an output tree with the root p to the descendants of p in the tree-of-trees relation.
  – $p2P^*$: a map from an element of the tree-of-trees relation to its descendants in the tree-of-trees relation.
  – PEl: the union of all elements of the removed sub-trees which are reachable from r. Here we use an additional parameter p. p is the root of the output tree to which r belongs. We have to use this extra parameter because we don't have a map from an element of an auxiliary tree relation to the root of the output tree which contains it.

```
removeSetTreeRelation3⟨Tree⟩($p_0$ : $ref!, p2r_0 : ref \to ref, R : Reg!,$
        $P : Reg!, r2R : ref \to Reg, c2r : ref \to ref, f_{PEL} : ref^2 \to ref,$
        $p2P : ref \to Reg!$)
{
Local variables:
        $\forall p : r2P^*[p] = P \cap r.\texttt{Desc}$;
        $\forall p : p2P^*[p] = P \cap p.\texttt{Desc}$;
        $\forall p, r : PEl[p, r] = \bigcup_{p' \in r2P[r]}^{f_{PEL}[p,.]} p'.\texttt{Desc} \cup \{p'\}$
        $\forall p : El[p] = (p.\texttt{Desc} \cup \{p\}) \setminus PEl[p, p2r_0[p]]$
        $\forall o : f_{inst}[o] = o.\texttt{instID}$;
        $\forall p : p2R_0[p] = El[p] \cap R$;
        $\forall p : p2R_0^+[p] = (El[p] \cap R) \cup \{\textbf{null}\}$;
        $\forall p : p2R[p] = p2R_0[p] \setminus \{p2r_0[p]\}$;
        $\forall p, r : rA_0[p, r] = \texttt{if}(r = \textbf{null})\ \texttt{then}\ p.\texttt{Anc}\ \texttt{else}\ r.\texttt{Anc} \cup \{r\}$;
        $\forall p, r, r' : rA[p, r, r'] = rA_0[p, r] \setminus rA_0[p, r']$;
Pre-conditions:
        $\forall p \in P$ : removeSetTreeRelation2
                $(p2r_0[p], p2R[p], p2P[p], r2R, c2r, f_{PEL}[p, .])$;
        $p_0.\texttt{instID} = p_0$;
        $P = \{p_0\} \cup p2P^*[p_0]$;
        $\forall p \in P : p2P[p] \subseteq p.\texttt{Desc}$;
        $\forall p \in P : p2P^*[p] = \left( \bigcup_{p' \in p2P[p]}^{f_{PEL}[p,.]} p2P^*[p'] \right) \cup p2P[r]$;
Transformations:
        $\langle R, \texttt{Child}, \mathbf{v}.\texttt{Child} \setminus r2P[\mathbf{v}] \rangle$;
        $\langle P, \texttt{parent}, \textbf{null} \rangle$;
        $\langle P\langle p \rangle.El, \texttt{root}, p \rangle$;
        $\langle P\langle p \rangle.El, Elements, El[p] \rangle$;
        $\langle \exists p, r, r' : p \in P \wedge r' \in p2R_0^+[p] \wedge r \in r2R[r'] \wedge \mathbf{v} \in rA[p, r, r'],$
                $\texttt{Desc}, \mathbf{v}.\texttt{Desc} \setminus PEl[p, r] \rangle$;
        $\langle P\langle p \rangle.El, \texttt{Anc}, \mathbf{v}.\texttt{Anc} \setminus p.\texttt{Anc} \rangle$;
Input instances: $\{p_0\}$
Frame:
        $\langle \mathbf{v}.\texttt{Desc} \cup \{\mathbf{v}\} \sharp R_{All}, \texttt{Desc} \rangle$;
        $\langle El[p_0], \texttt{Anc} \rangle$;
Measure: $P$
{
        removeSetTreeRelation2
                $(p2r_0[p_0], p2R[p_0], p2P[p_0], r2R, c2r, f_{PEL}[p_0, .])$;
        if($P \setminus \{p_0\} \neq \varnothing$) then {
        {
                $\Vert_{p \in p2P[p_0]}^{f_{inst}}$ removeSetTreeRelation3($p, p2r_0, p2R^*[p], p2P^*[p],$
                        $r2R, c2r, f_{PEL}[p, .], p2P$) measure $p2P^*[p_0] \overset{p_0}{\subset} P$;
        }
}
```

Figure B.6: `removeSetTreeRelation3` stereotype operation.

- El: the elements of the corresponding output tree. We define the elements of the output tree with the root p as the difference between the union of p and its descendants and the elements of the removed sub-trees.
- f_{inst}: is an auxiliary function which returns the instance identifier of the input parameter. We use it as a witness function to define the parallel composition.
- $p2R_0$, $p2R_0^+$, and $p2R$: defines R_0, R_0^+, and R for the output tree with the root p.
- rA_0 and rA: the definition is similar to the corresponding definitions of `removeSetTreeRelation2`. The only difference is that we define $rA_0[p, \mathbf{null}]$ as p.`Anc`. The idea behind this definition is that p is possibly removed from another output tree. In such a case the ancestors of p have to be removed from $rA[p, \mathbf{null}, p2r_0[p]]$.

- `Pre-conditions:`

17. for each output tree, all pre-conditions from `removeSetTreeRelation2` hold.
19. p_0 is an instance of the input tree.
20. an element of the tree-of-trees relation is either the root of the relation or a descendant of the root.
21. for each element p of the tree-of-trees relation, the children of p are descendants of p in the tree.
22. for each element p of the tree-of-trees relation, the descendants of p are equal to the union of the children of p and the union of the descendant of the children of p.

- `Transformations:` The transformation section is almost the same as for `removeSetTreeRelation2`. The only difference is that the rules for *Elements* and `Anc` are merged into single rules.
- `Input instances:` the instance identifier of the only input instance is p_0
- `Frame:` The frame section is almost the same as for `removeSetTreeRelation2`.
- `Measure:` as a measure we use the input parameter P.
- `Body:` The first statement of the body uses `removeSetTreeRelation2` to remove sub-trees from p_0. if $P \setminus \{p_0\} = \varnothing$ then we skip the recursive part of the operation. Otherwise we execute all recursive calls in

parallel. Each tree in the parallel composition is identified by p from $p2P[p_0]$. The values of the input parameters R and P in the recursive calls are equal to $p2R^*[p]$ and $p2P^*[p]$, respectively. Since the rest of the parameters are maps and the recursive calls rely on their properties on smaller domains we can reuse them without changes.

Bibliography

[1] S. B. Akers. Binary decision diagrams. *IEEE Trans. Computers*, 27(6):509–516, 1978.

[2] J. Avigad. Number theory and elementary arithmetic. *Philosophia Mathematica*, 11(3):257–284, 2003.

[3] A. Banerjee, M. Barnett, and D. Naumann. Boogie meets regions: A verification experience report. In *Verified Software: Theories, Tools, Experiments*, volume 5295 of *Lecture Notes in Computer Science*, pages 177–191. Springer Berlin / Heidelberg, 2008.

[4] A. Banerjee and D. A. Naumann. State based ownership, reentrance, and encapsulation. In *ECOOP*, pages 387–411, 2005.

[5] A. Banerjee, D. A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, pages 387–411, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] A. Banerjee, D. A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 387–411. Springer-Verlag, 2008.

[7] M. Barnett, M. Fähndrich, K. Rustan M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: The Spec# experience. *Communications of the ACM*, 54(6):81–91, June 2011.

[8] C. Barrett and C. Tinelli. CVC3. In *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, 2007.

[9] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[10] M. Benedikt, T. Reps, and M. Sagiv. A decidable logic for describing linked data structures. In *Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*, pages 641–641. Springer Berlin / Heidelberg, 1999.

[11] C. Benzmüller, L. C. Paulson, F. Theiss, and A. Fietzke. Leo-ii - a cooperative automatic theorem prover for classical higher-order logic (system description). In *Conference on Automated Deduction*, pages 162–170, 2008.

[12] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. Ohearn, and H. Yang. Shape analysis for composite data structures. In *In CAV*, pages 178–192. Springer, 2007.

[13] J. Berdine, C. Calcagno, and P. W. Ohearn. Smallfoot: Modular automatic assertion checking with separation logic. In *In International Symposium on Formal Methods for Components and Objects*, pages 115–137. Springer, 2005.

[14] J. Berdine, C. Calcagno, P. W. O'Hearn, and P. W. Ohearn. Symbolic execution with separation logic. In *In APLAS*, pages 52–68. Springer, 2005.

[15] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[16] E. Bevers and J. Lewi. Proof by consistency in conditional equational theories. In S. Kaplan and M. Okada, editors, *Conditional and Typed Rewriting Systems*, volume 516 of *Lecture Notes in Computer Science*, pages 194–205. Springer Berlin / Heidelberg, 1991.

[17] S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a c compiler front-end. In *14th International Symposium on Formal Methods*, pages 460–475, 2006.

[18] S. Böhme, K. Rustan M. Leino., and B. Wolff. HOL-Boogie an interactive prover for the boogie program-verifier. In *Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 150–166. Springer Berlin / Heidelberg, 2008.

[19] R. S. Boyer and J. S. Moore. *A computational logic.* ACM monograph series. Academic Press, 1979.

[20] R. A. Brualdi. *Introductory Combinatorics (4th Edition).* Pearson Prentice Hall, 2004.

[21] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The MathSAT 4SMT Solver. In *CAV*, pages 299–303, 2008.

[22] A. Bundy. The automation of proof by mathematical induction. In *Handbook of Automated Reasoning*, pages 845–911. 2001.

[23] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The oyster-clam system. In *Proceedings of the tenth international conference on Automated deduction*, CADE-10, pages 647–648, 1990.

[24] S. R. Buss. On herbrand's theorem. In *LCC*, pages 195–209, 1994.

[25] C. Calcagno and D. Distefano. Infer: an automatic program verifier for memory safety of C programs. In *Proceedings of the Third international conference on NASA Formal methods*, NFM'11, pages 459–465, Berlin, Heidelberg, 2011. Springer-Verlag.

[26] N. R. Cameron, S. Drossopoulou, J. Noble, and M. J. Smith. Multiple ownership. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 441–460, New York, NY, USA, 2007. ACM.

[27] C.-L. Chang and R. C.-T. Lee, editors. *Symbolic Logic, and Theorem Proving*. Academic Press, New York, 1971.

[28] Y. Cheon, G. Leavens, M. Sitaraman, and S. Edwards. Model variables: cleanly supporting abstraction in design by contract: Research articles. *Softw. Pract. Exper.*, 35(6):583–599, 2005.

[29] D. Clarke and T. Wrigstad. External uniqueness is unique enough. In L. Cardelli, editor, *ECOOP 2003 Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 59–67. Springer Berlin / Heidelberg, 2003.

[30] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 48–64. ACM, 1998.

[31] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A Practical System for Verifying Concurrent C. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, Berlin, Heidelberg, 2009. Springer-Verlag.

[32] H. Comon. Inductionless induction. In *Handbook of Automated Reasoning*, pages 913–962. 2001.

[33] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, third edition*. The MIT Press, 2009.

[34] B. Courcelle. *The expression of graph properties and graph transformations in monadic second-order logic*, pages 313–400. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[35] B. Courcelle. Graph structure and monadic second-order logic. January 2008.

[36] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '79, pages 269–282, New York, NY, USA, 1979. ACM.

[37] M. D'Agostino, D. M. Gabbay, R. Hähnle, and J. Posegga, editors. *Handbook of Tableau Methods*. Springer, 1999.

[38] Á. Darvas, F. Mehta, and A. Rudich. Efficient well-definedness checking. In A. Armando, P. Baumgartner, and G. Dowek, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, volume 5195 of *Lecture Notes in Computer Science*, pages 100–115. Springer-Verlag, 2008.

[39] L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *In Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[40] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52:365–473, May 2005.

[41] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In E. Ernst, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 4609 of *Lecture Notes in Computer Science*, pages 28–53. Springer-Verlag, 2007.

[42] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, October 2005.

[43] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18:453–457, August 1975.

[44] D. Distefano, P. W. O'Hearn, P. W. Ohearn, and H. Yang. A local shape analysis based on separation logic. In *In TACAS*, pages 287–302. Springer, 2006.

[45] D. Distefano and M. Parkinson. jStar: towards practical verification for java. In *Proceedings of the 23rd ACM SIGPLAN conference*

on Object-oriented programming systems languages and applications, OOPSLA '08, pages 213–226, New York, NY, USA, 2008. ACM.

[46] G. Dong and R. Topor. Incremental evaluation of datalog queries. In *Database Theory ICDT '92*, volume 646 of *Lecture Notes in Computer Science*, pages 282–296. Springer Berlin / Heidelberg, 1992.

[47] G. Z. Dong and J. W. Su. Incremental and decremental evaluation of transitive closure by first-order queries. *Information and Computation*, 120(1):101 – 106, 1995.

[48] B. Dutertre. Formal analysis of the priority ceiling protocol. In *Proceedings of the 21st IEEE conference on Real-time systems symposium*, RTSS'10, pages 151–160, Washington, DC, USA, 2000. IEEE Computer Society.

[49] M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 42, pages 337–350, New York, NY, USA, 2007. ACM.

[50] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, 1995.

[51] Y. Ge and L. Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 306–320. Springer-Verlag, 2009.

[52] G. Gentzen. The collected papers of gerhard gentzen. North-Holland, Amsterdam, 1969.

[53] K. Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für Mathematik*, 38:173–198, 1931.

[54] J. A. Goguen. How to prove algebraic inductive hypotheses without induction. In *Proceedings of the 5th Conference on Automated Deduction*, pages 356–373. Springer-Verlag, 1980.

[55] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, 2005.

[56] A. Gotsman and J. Berdine. Interprocedural shape analysis with separated heap abstractions. In *In SAS*, pages 240–260. Springer, 2006.

[57] E. Grädel, P. G. Kolaitis, and M. Y. Vardi. On the decision problem for two-variable first-order logic. *The Bulletin of Symbolic Logic*, 3(1), 1997.

[58] E. Gradel, M. Otto, and E. Rosen. Undecidability results on two-variable logics. In *STACS 97*, volume 1200 of *Lecture Notes in Computer Science*, pages 249–260. Springer Berlin / Heidelberg, 1997.

[59] E. Grandjean. Complexity of the first-order theory of almost all finite structures. *Information and Control*, 57(2-3):180 – 204, 1983.

[60] D. Harel, J. Tiuryn, and D. Kozen. *Dynamic Logic*. MIT Press, Cambridge, MA, USA, 2000.

[61] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *In EuroSys*, pages 177–190. ACM Press, 2006.

[62] M. Hennessy. *Semantics of programming languages - an elementary introduction using structural operational semantics*. Wiley, 1990.

[63] J. G. Henriksen, J. L. Jensen, M. E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Proceedings of the First International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 89–110. Springer-Verlag, 1995.

[64] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[65] G. Huet and J.-M. Hullot. Proofs by induction in equational theories with constructors. *Journal of Computer and System Sciences*, 25(2):239 – 266, 1982.

[66] D. Hutter and C. Sengler. Inka: The next generation. pages 288–292. Springer, 1996.

[67] N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In *In Computer Science Logic (CSL*, pages 160–174, 2004.

[68] N. Immerman, A. M. Rabinovich, T. W. Reps, S. Sagiv, and G. Yorsh. Verification via structure simulation. In *Computer Aided Verification*, pages 281–294, 2004.

[69] F. Ivancic, S. Sankaranarayanan, and C. Wang. Foreword: Special issue on numerical software verification. *Formal Methods in System Design*, 35(3):227–228, 2009.

[70] B. Jacobs and F. Piessens. The VeriFast Program Verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, 2008.

[71] B. Jacobs, J. Smans, and F. Piessens. Verifying the composite pattern using separation logic. In *Workshop on Specification and Verification of Component-Based Systems, Challenge Problem Track*, November 2008.

[72] J. L. Jensen, M. E. Jørgensen, M. I. Schwartzbach, and N. Klarlund. Automatic verification of pointer programs using monadic second-order logic. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, PLDI '97, pages 226–234. ACM, 1997.

[73] M. B. Jones. What really happened on Mars?, 1997. Available at: `http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Mars_Pathfinder.html`.

[74] J.-P. Jouannaud and E. Kounalis. Automatic proofs by induction in theories without constructors. *Information and Computation*, 82(1):1 – 33, 1989.

[75] K. Rustan M. Leino. This is Boogie 2. Technical report, Microsoft Research, 2008.

[76] K. Rustan M. Leino. Dafny: an automatic program verifier for functional correctness. In *Proceedings of the 16th international conference on Logic for programming, artificial intelligence, and reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.

[77] K. Rustan M. Leino and P. Müller. Object invariants in dynamic contexts. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004.

[78] K. Rustan M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *Foundations of Security Analysis and Design V*, volume 5705 of *Lecture Notes in Computer Science*, pages 195–222. Springer-Verlag, 2009.

[79] D. Kapur and H. Zhang. An overview of rewrite rule laboratory (rrl). *J. of Computer and Mathematics with Applications*, 29:91–114, 1995.

[80] I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM*, pages 268–283, 2006.

[81] I. T. Kassios. The dynamic frames theory. *Formal Asp. Comput.*, 23(3):267–288, 2011.

[82] M. Kaufmann and J. S. Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Transactions on Software Engineering*, 23:203–213, 1997.

[83] N. Klarlund and M. I. Schwartzbach. Graph types. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 196–205. ACM, 1993.

[84] N. Klarlund and M. I. Schwartzbach. Graphs and decidable transductions based on edge constraints (extended abstract). In *Proceedings of the 19th International Colloquium on Trees in Algebra and Programming*, pages 187–201. Springer-Verlag, 1994.

[85] K. Korovin. iProver – an instantiation-based theorem prover for first-order logic (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning, (IJCAR 2008)*, volume 5195 of *Lecture Notes in Computer Science*, pages 292–298. Springer, 2008.

[86] G. Kreisel. Mathematical Logic. In T. Saaty, editor, *Lectures on Modern Mathematics*, volume 3, pages 95 – 195. New York, 1965.

[87] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 17–32, New York, NY, USA, 2002.

[88] V. Kuncak and M. Rinard. Existential heap abstraction entailment is undecidable. In *Proceedings of the 10th international conference on Static analysis*, SAS'03, pages 418–438, Berlin, Heidelberg, 2003. Springer-Verlag.

[89] V. Kuncak and M. Rinard. Towards Efficient Satisfiability Checking for Boolean Algebra with Presburger Arithmetic. In *Automated Deduction CADE-21*, volume 4603 of *Lecture Notes in Computer Science*, pages 215–230. Springer Berlin / Heidelberg, 2007.

[90] S. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using smt solvers. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 171–182, New York, NY, USA, 2008.

[91] S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 115–126, New York, NY, USA, 2006. ACM.

[92] W. Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1:323–337, December 1992.

[93] D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *Proceedings of the 2nd World Congress on Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 806–809, Berlin, Heidelberg, 2009. Springer-Verlag.

[94] T. Lev-ami, N. Immerman, T. W. Reps, S. Sagiv, S. Srivastava, and G. Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In *Conference on Automated Deduction*, pages 99–115, 2005.

[95] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '00, pages 26–38, New York, NY, USA, 2000. ACM.

[96] B. Lutz. Error reporting for universe types with transfer. Master's thesis, ETH Zurich, Switzerland, 2008.

[97] R. Manevich, J. Berdine, B. Cook, G. Ramalingam, and M. Sagiv. Shape analysis by graph decomposition. In *Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'07, pages 3–18, Berlin, Heidelberg, 2007. Springer-Verlag.

[98] A. Meyer. Weak monadic second order theory of succesor is not elementary-recursive. In R. Parikh, editor, *Logic Colloquium*, volume 453 of *Lecture Notes in Mathematics*, pages 132–154. Springer Berlin / Heidelberg, 1975.

[99] A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 221–231. ACM, 2001.

[100] J. S. Moore, T. Lynch, and M. Kaufmann. A mechanically checked proof of the correctness of the kernel of the amd5k86 floating-point division algorithm. *IEEE Transactions on Computers*, 47, 1996.

[101] L. Moura and N. Bjørner. Engineering dpll(t) + saturation. In *Proceedings of the 4th international joint conference on Automated Reasoning*, IJCAR '08, pages 475–490. Springer-Verlag, 2008.

[102] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.

[103] P. Müller and A. Rudich. Formalization of ownership transfer in universe types. Technical Report 556, ETH Zurich, 2007.

[104] P. Müller and A. Rudich. Ownership transfer in Universe Types. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 461–478. ACM, 2007.

[105] D. R. Musser. On proving inductive properties of abstract data types. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '80, pages 154–162. ACM, 1980.

[106] A. Nanevski, A. Banerjee, and D. Garg. Verification of information flow and access control policies with dependent types. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, pages 165–179, Washington, DC, USA, 2011. IEEE Computer Society.

[107] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: dependent types for imperative programs. In *13th ACM SIGPLAN international conference on Functional programming*, pages 229–240, 2008.

[108] A. Nanevski, V. Vafeiadis, and J. Berdine. Structuring the verification of heap-manipulating programs. In *37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 261–274, 2010.

[109] D. A. Naumann and A. Banerjee. Dynamic boundaries: Information hiding by second order framing with first order assertions. In *European Symposium on Programming*, pages 2–22, 2010.

[110] G. C. Necula. Proof-carrying code. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Paris, France*, pages 106–119, Jan. 1997.

[111] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1:245–257, 1979.

[112] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53:937–977, November 2006.

[113] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[114] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of the 15th International Workshop on Computer Science Logic*, CSL '01, pages 1–19, London, UK, 2001. Springer-Verlag.

[115] M. Ottiger. Runtime support for generics and transfer in universe types. Master's thesis, ETH Zurich, Switzerland, 2008.

[116] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, Lecture Notes in Artificial Intelligence, pages 748–752. Springer-Verlag, jun 1992.

[117] A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership for generic java. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 311–324. ACM, 2006.

[118] M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Trans. Amer. Math. Soc.*, 141:1–35, 1969.

[119] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16:1467–1471, September 1994.

[120] T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. *ACM Trans. Program. Lang. Syst.*, 32:1–55, August 2010.

[121] A. Riazanov and A. Voronkov. The design and implementation of vampire. *AI Commun.*, 15:91–110, August 2002.

[122] N. Rinetzky, A. Poetzsch-Heffter, G. Ramalingam, M. Sagiv, and E. Yahav. Modular shape analysis for dynamically encapsulated programs. In R. De Nicola, editor, *Programming Languages and Systems*, volume 4421 of *Lecture Notes in Computer Science*, pages 220–236. Springer-Verlag, Berlin, Heidelberg, 2007.

[123] S. Rosenberg, A. Banerjee, and D. A. Naumann. Local reasoning and dynamic framing for the composite pattern and its clients. In *Proceedings of the Third international conference on Verified software: theories, tools, experiments*, VSTTE'10, pages 183–198, Berlin, Heidelberg, 2010. Springer-Verlag.

[124] A. Rudich, Á. Darvas, and P. Müller. Checking well-formedness of pure-method specifications. In J. Cuellar and T. Maibaum, editors, *Formal Methods (FM)*, volume 5014 of *Lecture Notes in Computer Science*, pages 68–83. Springer-Verlag, 2008.

[125] A. Rudich, Á. Darvas, and P. Müller. Checking well-formedness of pure-method specifications (full paper). Technical Report 588, ETH Zurich, 2008.

[126] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20:1–50, January 1998.

[127] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24:217–298, May 2002.

[128] A. Schaad. Inferring universe annotations in the presence of ownership transfer. Master's thesis, ETH Zurich, Switzerland, 2007.

[129] N. Schirmer. A Verification Environment for Sequential Imperative Programs in Isabelle/HOL. In *Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference*, Lecture Notes in Computer Science, pages 398–414. Springer, 2004.

[130] P. Schmitt, M. Ulbrich, and B. Weiß. Dynamic Frames in Java Dynamic Logic. In B. Beckert and C. Marche, editors, *Formal Verification of Object-Oriented Software*, volume 6528 of *Lecture Notes in Computer Science*, pages 138–152. Springer Berlin / Heidelberg, 2011.

[131] S. Schulz. E - a brainiac theorem prover. *AI Commun.*, 15:111–126, August 2002.

[132] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990.

[133] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames. In *Proceedings of the 10th ECOOP Workshop on Formal Techniques for Java-like Programs*, pages 1–12, July 2008.

[134] J. Smans, B. Jacobs, and F. Piessens. Vericool: An automatic verifier for a concurrent object-oriented language. In *Lecture Notes in Computer Science*, volume 5051/2008, pages 220–239. Springer, June 2008.

[135] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP 2009 - Object-oriented Programming, 23rd European Conference, Genova, Italy, July 6-10, 2009, Proceedings*, volume 5653, pages 148–172. Springer-Verlag, July 2009.

[136] J. Smans, B. Jacobs, and F. Piessens. Symbolic execution for implicit dynamic frames. Technical report, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, 2009.

[137] J. Smans, B. Jacobs, and F. Piessens. Heap-dependent expressions in separation logic. In *Formal Techniques for Distributed Systems*, volume 6117, pages 170–185. Springer-Verlag, June 2010.

[138] J. Smans, B. Jacobs, F. Piessens, and W. Schulte. An automatic verifier for Java-like programs based on dynamic frames. In *Lecture Notes in Computer Science*, volume 4961, pages 261–275. Springer, April 2008.

[139] Y. Takano. Implementing uniqueness and ownership transfer in the universe type system. Master's thesis, ETH Zurich, Switzerland, 2007.

[140] C. Tinelli and M. Harandi. A New Correctness Proof Of The Nelson-Oppen Combination Procedure. In *Frontiers of Combining Systems, volume 3 of Applied Logic Series*, pages 103–120. Kluwer Academic Publishers, 1996.

[141] B. A. Trakhtenbrot. Impossibility of an algorithm for the decision problem in finite classes. *Doklady Akademii Nauk SSSR*, 70:569–572, 1950.

[142] T. Tuerk. A separation logic framework for HOL. Technical Report UCAM-CL-TR-799, University of Cambridge, Computer Laboratory, June 2011.

[143] S. van Staden and C. Calcagno. Reasoning about multiple related abstractions with multistar. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 504–519, New York, NY, USA, 2010. ACM.

[144] C. Weidenbach, D. Dimova, A. Fietzke, M. Suda, and P. Wischnewski. Spass version 3.5. In R. A. Schmidt, editor, *Automated Deduction - CADE-22 : 22nd International Conference on Automated Deduction*, volume 5663 of *Lecture Notes in Artificial Intelligence*, pages 140–145, Montreal, Canada, August 2009. Springer.

[145] A. Wiles. Modular elliptic curves and Fermat's Last Theorem. *Annals of Mathematics*, 141(3):443–551, 1995.

[146] A. Wiles. Ring-Theoretic Properties of Certain Hecke Algebras. *Annals of Mathematics*, 141(3):553–572, 1995.

[147] C.-P. Wirth. History and future of implicit and inductionless induction: Beware the old jade and the zombie! In D. Hutter and W. Stephan, editors, *Mechanizing Mathematical Reasoning*, volume 2605 of *Lecture Notes in Computer Science*, pages 192–203. Springer, 2005.

[148] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *Proceedings of the 20th international conference on Computer Aided Verification*, CAV '08, pages 385–398, Berlin, Heidelberg, 2008. Springer-Verlag.

[149] G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *In 10th TACAS*, pages 530–545, 2004.

[150] K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 349–361, New York, NY, USA, 2008. ACM.

Curriculum Vitae

Personal Data

Name:	Arsenii Rudich
Date of Birth:	May 14, 1982
Nationality:	Ukraine
E-mail:	`arsenii.rudich@gmail.com`

Education

2005 - 2011	**Doctoral studies** ETH Zurich Chair of Programming Methodology Supervision: Prof. Dr. Peter Müller
2003 - 2005	**M.Sc. in computer science** National Taras Shevchenko University of Kyiv Department of Cybernetics Chair of Theoretical Cybernetics
1999 - 2003	**B.Sc. in applied mathematics** National Taras Shevchenko University of Kyiv Department of Cybernetics Chair of Computational Mathematics

Employment

2005 - 2011	Research and Teaching Assistant ETH Zurich
2003 - 2004	Researcher & Developer Materialise Inc.