Using stereotypes to verify complex heap structures in regional logic.

ETH Technical Report 651

Arsenii Rudich and Peter Müller

ETH Zurich, Switzerland

1 Introduction

Verification of complex heap structures is a challenging task. Typically, a description of heap structures requires intensive usage of transitive closure or similar constructions, e.g. recursive functions or inductive definitions. Such constructions are expressive enough to make writing of heap specifications fast and natural. On the other hand such specifications have several disadvantages:

- Because of their recursive nature it is hard to understand and modify them. Typically it is much simpler to write such a specification then understand what it actually means.
- Absence of a reusing mechanism for such specifications results in specifications duplications. For example, if part of a specified heap structure behaves like a tree or like a list, then such parts of specifications are typically copy-pasted from already existing specifications. Such copy-pastes slow down specifications writing and complicate their understanding.
- Direct using of transitive closure on real heap fields exposes implementation details. We would like have a way of specifying of heap structures which provides information hiding and avoids unnecessary restrictions on real heap structures.
- And finally, an ATP (automatic theorem prover) cannot adequately handle such constructs. Usually it is possible to describe such constructs, but their applications fairly often results in infinite looping.

To deal with these problems and limitations we introduce notion of *stereotype*. A stereotype describes an aspect of a heap structure from perspective of an object that participates in it. In contrast to an abstract class stereotype talks not only about an object or some specific set of objects, but about the whole heap. That is why it can be used locally to specify global properties. Such as stereotype is a purely abstract mechanism it can be easily reused by different heap structures. For the same reason stereotypes can be used as part of classes specifications to facilitate information hiding.

The remaining of the technical report is organized as follow: Section 2 explains notion of stereotype; in Section 3 we motivate necessity and define notion of stereotype operation; Section 4 describes which properties guaranties

stereotypes system invariant and which prove obligations have to be checked to guarantee soundness of the stereotype system invariant; Section 5 explains how relation between stereotypes and a real heap established using so called "glue" invariants; Section 6 describes how generalization of invariants can be used to build a verification technique on top of stereotypes.

2 Stereotype

From static (syntactical) perspective we define a stereotype as a ghost record equipped with stereotype invariants. By ghost we mean that all fields of a stereotype are used only for specifications purposes. Stereotype fields can have only primitive types, or special types *ref* and *Reg* which corresponds to an object reference and set of references. We assume that each stereotype has a special field `El` of type *Reg*. Stereotype invariants are formulas that universally quantified over reference type and additionally to standard operations can read values of stereotypes feeds.

The dynamical behavior of a stereotype is characterized by two notions: *stereotypes slice* and *stereotype instance*. Stereotype slice is extension of the heap by adding to each object stereotype's fields in such a way that their values satisfy stereotype's invariants. By adding special invariants that describes semantic of `El` it can be guarantee that stereotype fields from the slice form set of disjoint heap structures. Each of these structures we call stereotype instance. Because of disjointness property a stereotype instance can be identified by a participating object. Each object participates at exactly one stereotype instance from each stereotype slice. To summarize all above we can say that a stereotype slice is a set of disjoint stereotype instances. In general there can be several stereotypes slices for the same stereotype. Both number of stereotype slices and number of stereotype instances are unbounded.

Stereotype slice's definition raises a question why do we consider set of disjoint stereotype instances but not a single stereotype instance. Motivation for such a choice is that it is typical for data structure operations to merge and split data structures. For example: if an edge removed from a linked list then the list is split on two disjoint lists. If a reference from end of one list to begin of another list is added than two lists are merged in one. Our conclusion is that it is typical for data structures operations deal not with one data structure but with set of disjoint data structures. Someone can argue that another way to deal with such situation is to consider each stereotype instance separately. Disadvantage of such an approach is that in such a case we potentially need to deal with an unbounded set of data structures, which can be a problem for an ATP. For example, if we have list of size n and we want to sort it with merge sort then we get an unbounded number of lists during the algorithm execution. To avoid such a complication we treat all this lists together as one stereotype slice.

To enforce disjointness property we add the following invariants to stereotype invariants:

- $\forall o, o' : o' \in o.\texttt{El} \Rightarrow o'.\texttt{El} = o.\texttt{El}$

```
Tree{
    uniq root: ref;
    parent: ref;
    Children, Ancestors, Descendants: Reg;

    invariants ∀o, o′:
    root ≠ null
    o = root ⇔ o.parent = null
    root ∈ o.Ancestors
    o.El = root.Descendants ∪ {root}
    o′ ∈ o.Children ⇔ o′.parent = o
    o ≠ o′ ⇒ o.Children#o′.Children
    o ∉ o.Descendants
    o ∉ o.Ancestors
    o.Children ⊆ o.Descendants
    o′ ∈ o.Descendants ⇒ o.Children#o′.Descendants
    o′ ∈ o.Descendants ⇒ o ∈ o′.Ancestors
    o′ ∉ o.Descendants ⇒ o′.Ancestors#o.Descendants ∪ {o}
    o′ ∈ o.Descendants ⇒ o′.Ancestors \ o.Descendants = o.Ancestors ∪ {o}
    o′ ∈ o.Descendants ⇒ o′.Descendants ⊆ o.Descendants
    (o ≠ o′ ∧ o′ ∉ o.Descendants ∧ o ∉ o′.Descendants) ⇒
        ⇒ o′.Descendants#o.Descendants
    o′ ∈ o.Ancestors ⇒ o′.Ancestors ⊆ o.Ancestors
    o.parent = null ⇒ o.Ancestors = ∅
    o.parent ≠ null ⇒ o.Ancestors = o.parent.Ancestors ∪ {o.parent}
}
```

Fig. 1. Tree stereotype

- $\forall o, o' : o' \notin o.\texttt{El} \Rightarrow o'.\texttt{El} \# o.\texttt{El}$
- For each stereotype field f of type *ref* $\forall o : o.f \in o.\texttt{El}$
- For each stereotype field f of type *Reg* $\forall o : o.f \subseteq o.\texttt{El}$

Here we denote field reading as $o.f$, set disjointness as #, and quantification over non-null references as $\forall o$:.

To make stereotype description shorter we introduce type modifier `uniq`. If a field f of a stereotype declared with type modifier `uniq` then we add the following invariant to the stereotype invariants $\forall o, o' : o' \in o.\texttt{El} \Rightarrow o'.f = o.f$. Since value of such a field is the same for all object from the stereotype instance we omit receiver for such fields in stereotype invariants.

As example of stereotype let consider `Tree` stereotype (Figure 1). Here `root` is the root of the tree, `parent` is the parent of a tree element, `Children` inversion of `parent`, and `Descendants` and `Ancestors` are transitive closures of `Children` and `parent` respectively. `root` declared with `uniq`type modifier such as it is the same for all elements of a tree instances. We avoid explicit using of constructs like transitive closure and operations on sets of sets in stereotype invariants, such as this constructs can complicate work of ATP. As consequence we have

sound but not complete specifications in regional logic style. We assume that from pragmatic perspective completeness is desirable but not necessary property. It is good enough if stereotype invariants approximate properties of the data structure. For example, provided axioms of `Tree` stereotype are strong enough to describe topology of composite design pattern and priority inheritance protocol.

It can be easily seen that some of proposed invariants are redundant. Such redundant invariants can be proven as lemmas. Unfortunately some of these proofs require inductive reasoning, which is not feasible for an ATP. That is why we add such redundant properties explicitly as redundant invariants.

Another important observation is that in contrast to regional logic we allow using of recursive equations like $o.Ancestors = o.parent.Ancestors \cup \{o.parent\}$. Our experiments demonstrate that such constructions are feasible for an ATP.

To facilitate construction of more advanced stereotypes from simpler stereotypes we introduce notion of stereotype aggregation. As part of a stereotype declaration list of aggregated stereotypes can be mention. In such a case declared stereotype inherits all fields and invariants of aggregated stereotypes. To clarify terminology of inherited field they can be renamed in context of the declared stereotype. What is more important is that several fields of aggregated stereotypes can be merged into one during renaming. Trivial syntactical substitution can be applied to inherited invariants to adjust them.

As example of aggregation application we consider construction of a bidirectional list (Figure 2) from a unidirectional list (Figure 3). Unidirectional list is defined in terms of next element *next*, transitive closure of the next element $Next^*$, and last element of the list *end*. Bidirectional list aggregates two unidirectional lists Backward and Forward. Renaming establishes relation between fields of Backward and Forward and fields of bidirectional list. For example end of Backward corresponds to begin of bidirectional list. Additional invariants adjust relations between fields from different aggregated stereotypes. For example, $o.next \neq null \Rightarrow o.next.prev = o$ establish relation between *next* of Forward and Backward.

3 Stereotype operations

Up to now we considered stereotype slices like a static heap description. Such as heap changes during program execution the same should happen with stereotype slices. One possible way to do it is directly update stereotype fields of objects participating in updated stereotype instance. It is typical that such an update effects not only objects that directly participating in update, but many other objects reachable by transitive closure. For example if we want make an object o parent of an object o' then additionally to modifying *parent* field of o' we need to update *Ancestors* fields of all descendants of o' and *Descendants* field of all ancestors of o. Furthermore to preserve the stereotype invariant before the updated we need to check that o is not a descendant of o' and o' doesn't have another parent. As response to the shown complexity we introduce notion of stereotype operation. Stereotype operations:

```
UniList{
    uniq end: ref;
    next: ref;
    Next*: Reg;

    invariants ∀o, o′:
    o = end ⇔ o.next = null
    o.next = null ⇒ o.Next* = ∅
    o.next ≠ null ⇒ o.Next* = o.next.Next* ∪ {o.next}
    o ∉ o.Next*
    o ≠ end ⇒ end ∈ o.Next*
    o′ ∈ o.Next* ⇒ o′.Next* ⊆ o.Next*

}
```

Fig. 2. Unidirectional list stereotype

- guaranty stereotype's invariant preservation.
- specify operation's preconditions.
- group together semantically related transformations of stereotype fields.

A stereotype operation can read/write only stereotype fields and execute only other stereotype operations, but not methods of real classes. Such as stereotype operations change only stereotype fields they acts as ghost operations. Additionally to input parameters stereotype operations provide information about participating stereotype slices:

- Stereotype slices which the operation takes as input. We assume that they existed before begin of the operation execution and they exist after the end of the operation execution.
- Stereotypes slices that are added by the operation. They are disjoin from input stereotype slices. They are created at the begin of the operation execution and they exist after the end of the operation execution.
- Stereotype slices that are removed by the operation. They are a subset of the input stereotypes. It is assumed that they existed before begin of the operation execution and it is removed at the end of the operation execution.

Each participating stereotype slice has a unique name. If only one stereotype slice of a certain stereotype participates in the operation then name can be omitted.

As example of a stereotype operations let consider addition of a sub tree to a tree (Figure 4). `addSubTree` take as input one `Tree` stereotype slice, the name of the slice is omitted. Specification of the operation consists of precondition, postcondion, frame, and operation's body. Preconditions denoted by keyword *requires*. Preconditions of `addSubTree` requires that *tree* is not a descendent of *subTree* and *subTree* is the root. Preconditions are denoted by keyword *ensures*. Preconditions of `addSubTree` describes transformation performed during operation execution. Keyword $frame$ denote frame of the operation. Frame describes

```
List{
    uniq begin, end: ref;
    prev, next: ref;
    Prev*, Next*: Reg;

    aggregate:
        Backward, Forward: UniList

    rename:
        Forward.end → end
        Forward.next → next
        Forward.Next* → Next*
        Backward.end → begin
        Backward.next → prev
        Backward.Next* → Prev*

    invariants ∀o, o′:
    El = o.Prev* ∪ {o} ∪ o.Next*
    o.Next* # o.Prev*
    o.next ≠ null ⇒ o.next.prev = o
    o.prev ≠ null ⇒ o.prev.next = o

}
```

Fig. 3. Bidirectional list stereotype

set of locations (pair of a reference value and field name) that are possible affected during operation execution. We omit operations' bodies such as they are manly duplicate options' postconditions.

Additionally to stereotype operations we use side affect free functions. Mainly it is non recursive logical functions that are used to abbreviate typical expressions.

A typical example of such a function it is a singleton stereotype function. It describes values of fields for a singleton instance of a given stereotype. Figure 5 describes tree singleton function for a `Tree` stereotype. We assume that developer of a stereotype also describes singleton predicate. Because of the special meaning of the singleton predicate, the developer should prove that if it is valid for an object then stereotype's invariants of the corresponding stereotype invariants also valid for the object.

Another example of stereotype functions demonstrates how we can extract information useful for cyclic list from our general `List` stereotype. From informational perspective `List` stereotype has all necessary information to describe topology of cyclic list, but notions next element and previous elements requires some refinement. Figure 6 presents stereotype functions which refines this notions. This example demonstrates that if we assume bounded number of cycles

```
addSubTree⟨Tree⟩(tree: ref, subTree: ref)

requires tree ≠ null;
requires subTree ≠ null;
requires tree ≠ subTree;
requires tree ∉ subTree.Descendants;
requires subTree = subTree.root;

ensures subTree.parent = tree;
ensures ∀o : o ∈ old(tree.El) ∪ old(subTree.El) ⇒
    ⇒ o.El = old(tree.El) ∪ old(subTree.El);
ensures ∀o : o ∈ old(subTree.El) ⇒ o.root = tree.root;
ensures tree.Children = old(tree.Children) ∪ {subTree};
ensures ∀o : o ∈ tree.Ancestors ∪ {tree} ⇒ o.Descendants =;
    = old(o.Descendants) ∪ old(subTree.Descendants) ∪ {subTree};
ensures ∀o : o ∈ subTree.Descendants ∪ {subTree} ⇒ o.Ancestors =;
    = old(o.Ancestors) ∪ old(tree.Ancestors) ∪ {tree};

frame {⟨subTree, parent⟩}
      ∪{⟨tree, Children⟩}
      ∪(old(tree.El) ∪ old(subTree.El) × {El})
      ∪(old(subTree.El) × {root})
      ∪(tree.Ancestors ∪ {tree} × {Descendants})
      ∪(subTree.Descendants ∪ {subTree} × {Ancestors})

{
    ...
}
```

Fig. 4. `addSubTree` stereotype operation

we always can treat stereotype instances as acyclic structures and use special stereotype functions to receive specific for cyclic structures information.

Construction of stereotype operations requires significant efforts. One possible way to deal with this complexity it is to identify basic operations and construct more complex operations as composition of basic operations. Under an assumption about acyclicity of stereotype instances role of such operations can play addition and remove of an edge. We denote them as $\texttt{addEdge}(o, o', \texttt{St})$ and $\texttt{removeEdge}(o, o', \texttt{St})$, where o and o' are references, and `St` is a stereotype instance. We assume that developer of a stereotype always provides `addEdge` and `removeEdge` operations. `addSubTree` is an example of `addEdge` operation for `Tree` stereotype.

Such as this operations are associative, commutative, and idempotent we can generalize them up to addition remove of set of edges. We denote them as $\texttt{addSetEdge}(E, \texttt{St})$ and $\texttt{removeSetEdge}(E, \texttt{St})$, where o and o' are references, and `St` is a stereotype instance. Unfortunately using these operations in general

```
boolean isSingletonTree(tree: ref)

isSingletonTree(tree) = (tree.El = ∅ ∧ tree.root = tree ∧ tree.parent = null
∧tree.Ancestors = ∅ ∧ tree.Descendants = ∅ ∧ tree.Children = ∅)
```

Fig. 5. IsSingletonTree stereotype function

```
ref cycleNext(o: ref)
cycleNext(o) = if (o = o.end) then o.begin else o.next;

ref cyclePrev(o: ref)
cyclePrev(o) = if (o = o.begin) then o.end else o.prev;
```

Fig. 6. Refinement of notions next and previous for cyclic list

case requires dialing with families of sets which can be a problem for an ATP. But if set of ages have special form (for example, set of edges reachable by a transitive closure from one objet to another) then additional specifications can be provided to make these operations feasible for an ATP.

As example of construction of a more complex operation we consider stereotype operation `addSubTreeLoop` (Figure 7). It generalizes operation `addSubTreeLoop` on case when a *tree* is possibly a descendant of a *subTree*. The operation take as input two stereotype slices, a `Tree` and a `List`. Precondition guarantees that *tree* and each of its ancestors form a singleton instance of the `List` stereotype. In the case that *tree* is not a descendant of *subTree*, `addSubTree` is called. In other case all edges transitively reachable from *tree* to *subTree* (denoted as set E) are removed from the `Tree` slice and added to the `List` slice. The `List` slice are used to represent newly formed cycle in topology.

4 Stereotypes' system invariant

In the previous section we introduced the notion of stereotype operation. Execution of a stereotype operation can violate a stereotype invariant of some stereotype slice. To clarify question when and which stereotype invariants holds we describe stereotypes' system invariant. At the end of the section we sketch the soundness proof for system invariant. Soundness proof states that if source code satisfies certain syntactical restrictions and certain prove obligations hold then stereotypes' system invariant also holds.

But before we precede with the discussion of stereotypes' system invariant we need to establish a relation between stereotypes and classes. Similar to stereotype operations classes also can participate in stereotype slices. C `part` $\{\mathtt{St}_1, \ldots, \mathtt{St}_n\}$ denotes that class C participates in stereotype slices of stereotypes $St_1, \ldots, \mathtt{St}_n$, which means that all methods of the class C take as input these stereotype slices. Methods can declare added and removed slices. Removed slices have to

```
addSubTreeLoop⟨Tree, List⟩(tree: ref, subTree: ref)

requires tree ≠ null;
requires subTree ≠ null;
requires tree ≠ subTree;
requires subTree = subTree.root;
requires ∀o : o ∈ tree.Ancestors ∪ {tree} ⇒ isSingletonList(o);

{
var E : Reg × Reg;

if (tree ∉ subTree.Descendants)
then addSubTree(tree, subTree)
else {
     E := {⟨o, o.next⟩|o ∈ tree.Ancestors} \ {subTree};
     addSetEdge(E, List);
     removeSetEdge(E, Tree);
}}
```

Fig. 7. `addSubTreeLoop` stereotype operation

be disjoint from class's slices. If a class or a method declaration refers to several stereotype slices of the same stereotype then, similarly to stereotype operations, unique names are used to distinguish them. If there is an ambiguity at a method-/operation call which stereotype slices of calling method/operation correspond to stereotype slices of called method/operation correspond then it has to specified explicitly. For example, if there are two slices of list stereotype and we want to add an edge to one of them we have to specify which one we want to modify.

Now we can formulate stereotypes' system invariant. Stereotypes' system invariant states that if stereotype operations treated as atomic actions then stereotype invariants hold for all stereotype instances of all stereotype slides at each program point.

To guarantee soundness of the stereotypes' system invariant we put the following syntactical constraint on methods. A method can call stereotype operations and can read stereotype fields (for specifications purposes only), but can't directly write them. Mainly the constraint requires that all updates of stereotype fields from methods are done only via stereotype operations but not explicitly throw fields update.

The proof obligations for stereotype's system invariant are the following:

- It can be assumed that stereotype invariant for stereotypes mentioned in the verified stereotype operations' specification holds for all stereotype instances to which belong operation's input parameters after any stereotype operation call within the verified stereotype operation and at the begin of the operation.
- It has to be proven that stereotype invariant for stereotypes mentioned in the verified stereotype operations' specification holds for all stereotype in-

stances to which belong operation's input parameters before any stereotype operation call within the verified stereotype operation and at the end of the operation.
– It has to be proven that the verified stereotype operation eventually terminates.

Under the assumption that the syntactical constraints are satisfied and proof obligations hold, we can prove stereotype's system invariant. The proof is done by the induction on the number of calls. Sketch of the soundness proof:

– A method preserves stereotype invariants because it can't directly update stereotype fields and called stereotype operations preserve stereotype invariant because of induction hypothesis.
– In the stereotype operations we need take care only of stereotypes that are mentioned in the operation specifications and only of stereotype instances to which belong operation's input parameters. These invariants holds because of the prove obligations. Invariants of other stereotype slices can't be violated such as their field can't be read for syntactical rezones. Stereotype instances elements of which not include some of operation's input parameters can't be reached by the operation's expressions and as result can't be violated.
– If a called stereotype operation participates in some other stereotype (one which created temporally for some intermediate computations) then the called operation preserves the invariants because of induction hypnotize.

5 Glue invariant

Up to now we treated stereotype slices as purely abstract concept without any relation to real heap. To establish bridge between stereotypes and real heap we introduce the second layer of invariants, so called "glue" invariant. It relies on stereotype invariants and describes heap topology of specified data structures in terms of stereotype fields. The glue level consists of object states, local object states' invariants, and global object states' invariants. They have the following semantic:

– Object states describe different roles which an object can play in the data structure topology. They are similar to roles from role analysis.
– Local objects states' invariants describe relation between the object fields and stereotype fields for a single object. Usually such relations have the form of a condition if the object is in specific state then a field of the object equal to a stereotype field or belong to region which described by a region expression which possibly depends on stereotype fields.
– Global object states' invariants describe relation between states of different objects that participate in the data structure. Global object states' invariants are described in terms of object states and stereotype fields, but not in terms of class fields. These invariants are universally quantified over heap objects and are used to describe global properties of the data structure.

During execution of methods of objects which are involved in a heap structure the glue invariant can be temporally violated. For example, we can't avoid temporal violation of the glue invariant during addition of an element to a double linked list. On other hand, because of the flexibility provided by object states, without loss of flexibility we can require that the glue invariant holds in all visible program state (before a method call and at the end of a method). In situations when one method breaks a "glue" invariant and calls on other one to restore it, we can add one more state which describes how the topology before second method call differs from other states. After this the first method doesn't break the "glue" invariant but changes objet's state. Standard approach to verification of invariants with visible state semantic can be used to verify "glue" invariants.

Nevertheless, we introduce `expose` construct that explicitly indicates in which program points state invariants of which objets can be violated. Such additional information makes verification of glue invariants feasible for an ATP. Alternatively an ATP can infer this information, but usually it requires programmer to prove interactively a part of the prove obligations. Additional advantage of the `expose` construct is that heap topology modifications become explicit. It significantly increases readability of the code.

As parameter `expose` construct takes sets of objects that are being exposed. Sets of exposed objects can be guarded by conditions, in such a case the set is exposed if and only if the condition holds. If objects change state after execution of the `expose` block then a new state is provided after a colon. To express semantics of the `expose` construct we introduce additional implicit object state $\mathtt{st}_{Broken}$. If an object state is $\mathtt{st}_{Broken}$ then it is excluded from the system glue invariant. This property is expressed via the following instrumentation of a glue invariant: $\forall o_1, \ldots, o_n : \phi$ is replaced by $\forall o_1, \ldots, o_n : (\bigwedge_{i=1}^{n} o_i.\mathtt{st} \neq \mathtt{st}_{Broken}) \Rightarrow \phi$. The semantics of the `expose` statements can be expressed via if statements and assignments as shown in Figure 8.We assume that all state changes are done explicitly within an `expose` block.

The glue system invariant states that at any program point outside of an outermost `expose` block all allocated objects are not broken and glue invariants hold for all objects. To guarantee soundness of our approach we put syntactical and semantic restrictions on a program. There are two syntactical restrictions. Such as state of a freshly allocated object is $\mathtt{st}_{Broken}$ we require that any constructor starts from `expose`($this$) block. Such as inside of an `expose` block some objects can be in a broken state we forbid method calls inside of an `expose` block.

We need to check two kinds of proof obligations to guarantee soundness of the system glue invariant. To generate proof obligations we need the footprint of glue invariants and the set of currently exposed objects. The first one can be automatically extracted from glue invariants (because of their special structure). The second one can be computed from parameters of `expose` constructions. Then the first kind of proof obligations which has to be checked is that if field f of object o is written then for all objects o', such that $\langle o, f \rangle$ belongs to frame of glue invariant of o', o' belongs to the set of exposed objects. The second proof obli-

```
expose(
  if (cond_1) then Set_1 : st_1
  ..........................
  else if (cond_{n-1}) then Set_{n-1} : st_{n-1}
  else  Set_n : st_n
){
    Statement
}
```

⇓

```
var b_1,...,b_{n-1} : boolean;
var r_1,...,r_n : region;
b_1 := cond_1;
..........................
b_{n-1} := cond_{n-1};
r_1 := Set_1;
..........................
r_n := Set_n;
if (b_1) then foreach(o ∈ r_1, o.st := st_Broken);
..........................
else if (b_{n-1}) then foreach(o ∈ r_{n-1}, o.st := st_Broken);
else  foreach(o ∈ r_n, o.st := st_Broken);
Statement;
if (b_1) then foreach(o ∈ r_1, o.st := st_1);
..........................
else if (b_{n-1}) then foreach(o ∈ r_{n-1}, o.st := st_{n-1});
else  foreach(o ∈ r_n, o.st := st_n);
}
```

Fig. 8. Semantics of the `expose` construct.

gations is that at the end of the expose statement, glue invariants of all exposed objects have to hold. It can be proven by the induction on the method calls and size of the methods that stated proof obligations and syntactical restrictions are sufficient enough to guarantee glue system invariant.

Universal quantification over all allocated objects results in necessity to check global property. It violates modularity of the approach. Any program statement can potentially violate any "glue" invariant. But such as we know topology of the data structure we can solve this problem by adding a syntactical constraint that only objet of predefined classes (classes that participate in the data structure) can modify fields that participate in the data structure through direct field updates. It gives a natural way to define a notion of data structure (or topology of design pattern) and describe its boundaries in a modular way (a module defines a set of classes which form a data structure). If the data structure consists only of instances of one class it is enough to declare fields that forms the data structure as private.

As example of "glue" invariant construction let's consider the description of the topology of the process inheritance protocol (PIP) see Figure 9. The topology of PIP consist of instances of class `Node` that have a field *blockedBy*. If the field *blockedBy* of a node o is equal to *null* then we call node o free. The protocol supports two operations, release and acquire. If a free node o blocked node o' then release operation make node o' free. If a free node o acquire a node o' then in case if o' is free o block o' in other case (when o' is not free) o' block o.

There are mainly two possibilities for shape of PIP topology. The fist one is the deadlock-free case. In this case the topology of PIP is a tree formed by *blockedBy* fields. Acquire and release correspond to addition and removal of a subtree. But if a node acquires one of its descendants then a deadlock created. In this case the topology of PIP forms a so called sun shape. A sun shape consists of a set of trees. Roots of the trees forms a cycle of deadlocked nodes. Such topology naturally suggests composition of two shapes `Tree` and `List`. We can see that each node can play one of two roles in topology, namely either it is in a deadlock or not. To formalize previous observation we introduce two object states, deadlocked (denoted as $\texttt{st}_D$) and locked (denoted as $\texttt{st}_L$). Their semantics are described by the following local object state invariants:

- $\forall\ o : o.\texttt{st} = \texttt{st}_L \Leftrightarrow o.blockedBy = o.\texttt{Tree}.parent$ If an object is in the locked state then its field *blockedBy* plays the role of a parent in the tree stereotype slice.
- $\forall\ o : o.\texttt{st} = \texttt{st}_D \Leftrightarrow o.blockedBy = cycleNext(o)$ If an object is in the locked state then its field *blockedBy* plays the role of a next element in the list stereotype slice.

And finally to describe topology of PIP using introduced object states and stereotype slices we use the following global object state invariants:

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

```
class Node⟨Tree, List⟩{
  blockedBy : Node;
  st : {st_L, st_D};

  void release(subTree: Node)
   requires subTree ≠ null;
   requires this.Tree.root = this;
   requires subTree ∈ this.Tree.Children;

   stereotype: removeEdge(subTree, this, Tree);
  {
    expose(tree){
      removeEdge(subTree, this, Tree);
      subTree.blockedBy = null;
    }
  }

  void acquire(subTree: Node)
   requires subTree ≠ null;
   requires this.Tree.root = this;

   stereotype: subTree.Tree.parent = null ⇒ addEdge(subTree, this, Tree);
   stereotype: subTree.Tree.parent ≠ null ⇒ addSubTreeLoop(tree, subTree);

   state: subTree.Tree.parent ≠ null ∧ tree ∈ subTree.Tree.Descendants ⇒
     ⇒ ∀ o ∈ tree.Tree.Ancestors ∪ {tree} : o.st = st_D;
  {
   if (subTree.blockedBy = null)
   then{
     expose(tree){
       addEdge(subTree, this, Tree);
       subTree.blockedBy = this;
     }
   } else {
     expose(
       if (tree ∈ subTree.Tree.Descendants)
       then tree.Tree.Ancestors ∪ {tree} : st_D
       else subTree
     ){
       addSubTreeLoop(tree, subTree);
       tree.blockedBy = subTree;
     }
   }
  }
}
```

Fig. 9. Topological aspect of class `Node` of priority inheritance protocol.

– $\forall\ o, o' : o' \in o.\texttt{Tree}.Descendants \Rightarrow o'.\texttt{st} = \texttt{st}_L$ If an objet is a descendant of another object then it is in state locked.
– $\forall\ o, o' : o.\texttt{st} = \texttt{st}_D \wedge o' \in o.\texttt{List.El} \Rightarrow o'.\texttt{st} = \texttt{st}_D$ If an object o' participates in the same instance of the list with object o and the state of o is deadlock then the state of o' is also deadlock.

We need to prove that the "glue" invariants are preserved by all class methods. Additionally to pre/post conditions we use stereotype specifications (denoted by keyword "stereotype") and state changes specifications (denoted by keyword "state"). Stereotype specifications have the form of a sequence of stereotype operations possibly guarded by a condition. Stereotype specifications use stereotype operations to describe changes of stereotype fields and frame of these changes. A weakest precondition calculus can be used to transform stereotype specification to a normal logical formula (since postconditions of stereotype operations have the form of equations, we mainly use substitution for this transformation). Normal postconditions can refer to the stereotype fields but normal frame can't. State changes specifications describes new states of object which states are changed during the operation execution. Normal postconditions can refer to the object state but normal frame can't.

As it was mentioned above, class `Node` has two methods `release` and `acquire`. Method `release` use stereotype operation `removeEdge` to remove the edge from `Tree` stereotype slice. Method `release` temporally exposes *tree* and preserves all objects' states. There are two main possibilities for behavior of method `acquire`. If *subTree* is free (or what is the same *subTree* is a root of the tree instances) then `removeEdge` used to add the edge to `Tree` stereotype slice. In this case *subTree* is temporally exposed and all objects' states are preserved. In the other case, the operation `addSubTreeLoop` is used to rebuild `Tree` and `List` stereotype. This reconstruction involves creation of a loop described by the list stereotype in case if *tree* is descendant of *subTree*. In the last case states of *tree* and all objects reachable from *tree* to *subTree* by parent relation are temporally exposed and changed to the deadlocked state.

You can see that methods specification ignore changes of a real heap structure. This underspecification guaranties information hiding. The client code shouldn't rely on the actual implementation. It should use only stereotype fields and object states. Stereotype invariants and global object states' invariants (these invariants have to hold after each method execution) provide all necessary information about their semantic. On the other hand, during verification of class methods we can rely on local object states' invariants. They provide enough information to restore topology of real heap.

6 Invariant's state

In this chapter we consider how to specify and verify object invariants. It is typical that objet's invariants depends of fields of several objects. It is possible that these dependencies change dynamically and the number of objects on which

an invariant depends is unbounded. To deal with this complexity we build object invariants on top of stereotypes and glue invariants. The last two describe the topology of the heap structures of the verified classes.

We demonstrate our approach to object invariants verification using the PIP example. We build PIP object invariants on top of topology properties which we specified in the previous section. The goal of the PIP protocol is to guarantee specific properties of nodes' priorities. We will describe these properties later. There are three additional fields in the `Node` class that are used to express priority related properties, see Figure 10. `currentPriority` is the current priority of a node. It can increase and decrease during program execution. The initial value of `currentPriority` is equal to `defaultPriority`. `defaultPriority` is constant for an object. `priorities` is a multiset that contains the current priorities of the nodes blocked by this node.

```
class Node⟨Tree,List⟩{
  blockedBy: Node;
  st:{st_L,st_D};
  defaultPriority: int;
  currentPriority: int;
  priorities: MultiSet⟨int⟩;
}
```

Fig. 10. Class `Node` of priority inheritance protocol.

To specify and verify invariants we need to explicitly specify the set of the objects on which an invariant depends. We can specify these sets in terms of stereotype elements and object states. For example, to specify the PIP example we need the inversion and the transitive closure of `blockedBy` field see Figure 11. We denote as:

- $\texttt{blockedBy}^{-1}$ inversion of `blockedBy` field.
- $\texttt{blockedBy}^{*}$ transitive and reflexive closure of `blockedBy` field.
- $\texttt{blockedBy}^{+}$ transitive and irreflexive closure of `blockedBy` field.

We can prove characteristic properties of these predicates. For example, an ATP can prove that $\forall\ o, o' : o'.blocekdBy = o \Leftrightarrow o' \in \texttt{blockedBy}^{-1}(o)$. For the transitive closure we can prove that $\forall\ o : \texttt{blockedBy}^{*}(o) \supseteq o.blockedBy^{*}$ but not the other way round because the last one statement can be expressed only via existential quantification which is not feasible for an ATP. Nevertheless it is not a problem since we can prove practically valuable properties of the predicate $\texttt{blockedBy}^{*}$ using stereotype invariants.

As you can see all these predicate definitions have local nature. We don't need universal quantification to express them. Mainly it is because we implicitly outsource these quantifiers to stereotype invariants. In such a way we can, as we stated earlier, use stereotypes to express global properties in a local way.

```
Reg blockedBy^{-1}(o: ref)
blockedBy^{-1}(o) = if (o.st = st_L) then o.Tree.Children
                    else o.Tree.Children ∪ {o.List.prev};

Reg blockedBy^{*}(o: ref)
blockedBy^{*}(o) = o.Tree.Ancestors ∪ o.Tree.root.List.El ∪ {o};

Reg blockedBy^{+}(o: ref)
blockedBy^{+}(o) = blockedBy^{*}(o) \ {o};
```

Fig. 11. `blocedBy` predicates

To facilitate stereotype-based verification, we generalized the notion of object invariant to a notion of invariant state. In ownership-based verification, it is assumed that an object invariant depends only on the part of the heap which is owned (encapsulated) by the object. This guarantees that the invariant can be broken only by the object. From the practical perspective this restriction is too strong. It is typical that the invariants depend on heap locations that can be changed by other objects. In this case, the invariant is broken and essential information about the object state is lost. It is not clear which methods need to be involved to restore the object invariant. One possible way to deal with this problem and make objects suitable for such external heap changes is to introduce invariant states. It is usually assumed that an invariant is either valid or invalid, which can be treated like that the invariants state takes values from the Boolean domain. Instead, we assume that the class developer provides a description of the invariant states values domain. Together with the domain description, the developer should provide and prove transitions rules. These rules describe how the objects invariant state changes as a reaction to external heap changes. The rules formalize the developers expectations about how the environment of the object can be changed during the program execution. If the provided rules are detailed enough, they can be used to restore the object invariant after external changes.

To specify priority related behavior of PIP we introduce three object invariants in Figure 12. `invDefaultPriority` invariant specifies properties of the default priority. This invariant is established when an object is created and holds during the object's lifetime. `invPriorities` invariant specifies the semantic of the *priorities* multiset. Its state space is the set of pairs of natural numbers. Essentially the `invPriorities` invariant describes how to construct the multiset of priorities of nodes blocked by the this node. It states that to receive the desired multiset it is enough to add `to` (if $\texttt{to} \neq 0$) to and remove `from` (if $\texttt{from} \neq 0$) from `priorities`. The `currentPriority` invariant describes the relation between the current priority of the object and the current priorities of objects blocked by the object. `currentPriority` contains meaningful value only if the invariant state of the invariant `invPriorities` is $\langle 0, 0 \rangle$.

```
invariant invPriorities
state space: ℕ × ℕ
⟨from, to⟩ ⇒
    ∀i ∈ ℕ⁺ \ {from, to} : priorities[i] =
        |{o ∈ blockedBy⁻¹(this) : o.currentPriority = i}|
    from ≠ 0 ⇒ priorities[from] =
        |{o ∈ blockedBy⁻¹(this) : o.currentPriority = from}| + 1
    to ≠ 0 ⇒ priorities[to] =
        |{o ∈ blockedBy⁻¹(this) : o.currentPriority = to}| − 1

invariant invCurrentPriority
state space: Boolean
true ⇒ currentPriority = max(priorities ∪ {defaultPriority})

invariant invDefaultPriority
state space: Boolean
true ⇒ defaultPriority > 0
```

Fig. 12. `Node` invariants

As we can see both `invDefaultPriority` and `currentPriority` invariants are independent from fields of other objects. As a consequence these invariants can be affected only via explicit writing to fields of the object. It makes the tracking of invariants states fairly simple. A method can affect these invariants of an object o only if the object o is explicitly mentioned in the write effect closure of the method. On the other hand the `invPriorities` invariant depends from values of fields of a possibly unbound number of objects. As a result, even if an object o isn't mentioned in a method's write effect closure, an execution of the method can affect the value of the state of the `invPriorities` invariant. To facilitate verification of such invariants we expect the user to provide transition rules that express how the object state evolves by modifications of fields of other objects. The transition rules for `invPriorities` invariant are provided on Figure 13.

Essentially the transition rules describe a relation between the state of the program invariant in initial and final program points. A transition rule description consists of the following components:

- `Name` - name of the rule. Used for references to the rule.
- `Initial state` - state of the invariant in the initial program point.
- `Frame` - which part of the heap should be preserved between initial and final program points.
- `Heap conditions` - relation between heaps at initial and final program point. Usually describe how the part of the heap that is relevant to the invariant and doesn't belong to the frame was changed.
- `Final state` - state of the invariant in the final program point.

There are three kinds of transition rules:

Name: Frame
Initial state: $\langle from, to \rangle$
Frame: $\{\langle this, priorities \rangle, \langle this, \texttt{blockedBy}^{-1} \rangle,$
$\texttt{blockedBy}^{-1}(this) \times \{currentPriority\}\}$
Heap conditions: $true$
Final state: $\langle from, to \rangle$

Name: Remove a child
Initial state: $\langle 0, 0 \rangle$
Frame: $\{\langle this, priorities \rangle, (\texttt{blockedBy}^{-1}(this) \setminus \{child\}) \times \{currentPriority\}\}$
Heap conditions:
$child \in old(\texttt{blockedBy}^{-1}(this))$
$\texttt{blockedBy}^{-1}(this) = old(\texttt{blockedBy}^{-1}(this)) \setminus \{child\}$
Final state: $\langle old(child.currentPriority), 0 \rangle$

Name: Add a child
Initial state: $\langle 0, 0 \rangle$
Frame: $\{\langle this, priorities \rangle, (\texttt{blockedBy}^{-1}(this) \setminus \{child\}) \times \{currentPriority\}\}$
Heap conditions:
$child \notin old(\texttt{blockedBy}^{-1}(this))$
$\texttt{blockedBy}^{-1}(this) = old(\texttt{blockedBy}^{-1}(this)) \cup \{child\}$
Final state: $\langle 0, child.currentPriority \rangle$

Name: Change current priority of a child
Initial state: $\langle 0, 0 \rangle$
Frame: $\{\langle this, priorities \rangle, \langle this, \texttt{blockedBy}^{-1} \rangle,$
$(\texttt{blockedBy}^{-1}(this) \setminus \{child\}) \times \{currentPriority\}\}$
Heap conditions: $child \in old(\texttt{blockedBy}^{-1}(this))$
Final state: $\langle old(child.currentPriority), child.currentPriority \rangle$

Name: Update from
Initial state: $\langle from, to \rangle$
Frame: $\{\langle this, \texttt{blockedBy}^{-1} \rangle, \texttt{blockedBy}^{-1}(this) \times \{currentPriority\}\}$
Heap conditions: $priorities = old(priorities) \setminus \{from\}$
Final state: $\langle 0, to \rangle$

Name: Update to
Initial state: $\langle from, to \rangle$
Frame: $\{\langle this, \texttt{blockedBy}^{-1} \rangle, \texttt{blockedBy}^{-1}(this) \times \{currentPriority\}\}$
Heap conditions: $priorities = old(priorities) \cup \{to\}$
Final state: $\langle from, 0 \rangle$

Name: Normalize state
Initial state: $\langle a, a \rangle$
Frame: $\varnothing$
Heap conditions: $\$Heap = old(\$Heap)$
Final state: $\langle 0, 0 \rangle$

Fig. 13. `invPriorities` transition rules

– Frame rules
– State update rules.
– Normalization rules.

Frame rules describe conditions under which the value of the invariant state is preserved. Frame rules have the same value for initial and final states. Since invariants definitions can include predicates there can be several frame rules. But even if they are equivalent it still can be useful to have all of them. Such a redundant information can make more proofs feasible for an ATP. In our case we have only one frame rule, the first one. Also it is possible that there are different frame rules for different states since they rely on different parts of the heap.

State update rules describe how modifications of locations that are relevant to the invariant change the invariant's state. In our case rules `Remove a child` and `Add a child` describe the effect of the $\texttt{blockedBy}^{-1}(this)$ modification. `Change current priority of a child` describes how a change of the current priority of a node blocked by this affects the invariant state. The `Update from` and the `Update to` describe which modifications of `priorities` multiset can restore invariant state back to $\langle 0, 0\rangle$.

It is possible that different invariant states are equivalent, namely they give the same guaranty. For example invariant state $\langle a, a\rangle$ $(a > 0)$ of the `invPriorities` invariant states that we should add a and remove a from the multiset `priorities` to receive a multiset of current priorities of objects blocked by the object. The result of this operations is multiset `priorities`.We can see that $\langle a, a\rangle$ is equivalent to $\langle 0, 0\rangle$. To formalize such kind of equivalence we use normalization rules. Normalization rules are applied to a single program point, as a consequence they don't need a frame property, and the heap property states that the old heap is equal to the new one. The `invPriorities` invariant has one normalization rule, the last one on Figure 13.

To facilitate tracking of an object invariant states we make all updates of invariant states explicit. For this purposes we introduce the `exposeInv` construct. In general `exposeInv` plays the same role for object invariants that `expose` plays for glue invariants. The same approach that we use to verify glue invariants can be used to verify object invariants. In contrast to glue invariant we don't require an object invariant always to be in a consistent state. As consequence we can call methods from bodies of an `exposeInv` construct.

Let's consider the specification of PIP. To increase readability we drop all topology related specifications. Since glue invariants are independent from topology invariants we can implement verification as two-pass procedure. First we verify the topology and then assume topological properties during verification of object invariants.

In addition to pure topological version of PIP we need one more method to update the value of current priority of PIP nodes, see Figure 14. You can see that we introduce one more specification construct `invariants update`. These construct specify states of which invariants are changed during method execution. We need this information for purposes of modularity. These specifications can be verified by checking invariants of which objects are mentioned as parame-

```
void updatePriorities(from: int, to: int)

requires ∀o ∈ blockedBy+(this) : o.invPriorities = ⟨0,0⟩∧
           o.invCurrentPriority ∧ o.invDefaultPriority;
requires this.invPriorities = ⟨from,to⟩ ∧ this.invDefaultPriority
invariants update: this.invPriorities : ⟨0,0⟩
invariants update: this.invCurrentPriority : True;
frame: blockedBy*(this) × {currentPriority, priorities}

 {
  var oldCurrentPriority int;

  oldCurrentPriority = currentPriority;

  exposeInv(this.invCurrentPriority: b → True){
    if (from > 0) then
      exposeInv(this.invPriorities: ⟨a,b⟩ → ⟨0,b⟩){
        priorities = priorities \ {from};
      };
    if (to > 0) then
      exposeInv(this.invPriorities: ⟨0,b⟩ → ⟨0,0⟩){
        priorities = priorities ∪ {to};
      };

    exposeInv(
      if(blockedBy ≠ null) then blockedBy.invPriorities:
        ⟨0,0⟩ → ⟨old(currentPriority), currentPriority⟩
          by rule⟨Change current priority of a child⟩;
    ){
      currentPriority = max(defaultPriority, max(priorities));
    }
  }

  if(blockedBy ≠ null and oldCurrentPriority ≠ CurrentPriority)
    then blockedBy.updatePriorities(oldCurrentPriority, currentPriority);
  }

  exposeInv(
    if(blockedBy ≠ null and oldCurrentPriority = CurrentPriority)
      then blockedBy.invPriorities:
        ⟨currentPriority, currentPriority⟩ → ⟨0,0⟩
          by rule⟨Normalize state⟩;
  );

}
```

Fig. 14. Update priorities method of class `Node` of priority inheritance protocol.

```
void release(subTree: Node)

requires ∀o ∈ blockedBy*(subTree) : o.invPriorities = ⟨0, 0⟩∧
              o.invCurrentPriority ∧ o.invDefaultPriority;
{
  exposeInv(
    this.invPriorities: ⟨0, 0⟩ → ⟨subTree.currentPriority, 0⟩
      by rule⟨Remove a child⟩;
  ){
    subTree.blockedBy = null;
  }
  if (subTree.currentPriority ≠ 0)
    then updatePriorities(subTree.currentPriority, 0);
}

void acquire(subTree: Node)

requires ∀o ∈ blockedBy*(this) ∪ blockedBy*(subTree) :
o.invPriorities = ⟨0, 0⟩ ∧ o.invCurrentPriority ∧ o.invDefaultPriority;

{
 if (subTree.blockedBy = null)
 then{
  exposeInv(
    this.invPriorities: ⟨0, 0⟩ → ⟨0, subTree.currentPriority⟩
      by rule⟨Add a child⟩;
  ){
   subTree.blockedBy = this;
  }
   if (subTree.currentPriority ≠ 0)
     then updatePriorities(0, subTree.currentPriority);
 } else {
   exposeInv(
    subTree.invPriorities: ⟨0, 0⟩ → ⟨0, this.currentPriority⟩
      by rule⟨Add a child⟩;
  ){
   this.blockedBy = subTree;
  }
   if (currentPriority ≠ 0)
     then subTree.updatePriorities(0, this.currentPriority);
 }
}
```

Fig. 15. Release and acquire methods of class `Node` of priority inheritance protocol.

ters of an `exposeInv` construct. Method `updatePriorities` changes only states of object invariants `invPriorities` of object *this* to $\langle 0, 0\rangle$ and states of object invariants `invCurrentPriority` of object *this* to T*rue*.

At the begin of the method we expose *this*.`invCurrentPriority` invariant. New value of the invariant state is *True*. It is consistent with the `invariants update` specifications. This exposure enables a change of current priority `priorities` multiset inside of the nested `exposeInv` blocks. The first two expose blocks change the state of invariant `invPriorities` from state $\langle a, b\rangle$ to state $\langle 0, b\rangle$ and then to state $\langle 0, 0\rangle$. The third one describes how a change of the current priority affects the value of the `invPriorities` invariant state of the parent object. At the end of the method we can see an example of an application of a normalization rule. In case if after all transformations the value of current priority of the `this` object is preserved then the value of the `invPriorities` invariant state of the parent object is $\langle currentPriority, currentPriority\rangle$, which is equivalent to $\langle 0, 0\rangle$.

In method `release` *subTree* is removed from the set of nodes blocked by *this*. It affects the invariant state of the *this*.`invPriorities` invariant. State of the invariant is changed from $\langle 0, 0\rangle$ to $\langle subTree.currentPriority, 0\rangle$ nevertheless it doesn't mention in the `invariants update` specification. The reason for this is that the execution of the method `updatePriorities` restores the initial state of the invariant and its temporally changes doesn't visitable outside of the method. Method `acquire` is similar to method `release`. If *subTree* is free (doesn't have a parent) then *subTree* is added to *this* children and *this*.`invPriorities` is exposed. In other case *this* is added to *subTree* children and *subTree*.`invPriorities` is exposed. Similarly to `release` method absence of `invariants update` specifications express that values of object invariant states are preserved.