Formal Verification of Recursive Predicates

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der Fakultät für Informatik

der Universität Fridericana zu Karlsruhe (TH)

genehmigte

Dissertation

von

Richard Bubel

aus Erlangen

Tag der mündlichen Prüfung: 29.06.2007

Erster Gutachter: Prof. Dr. P. H. Schmitt, Universität Karlsruhe (TH)

Zweiter Gutachter: Prof. Dr. U. Furbach, Universität Koblenz-Landau

Contents

Acknowledgement

In the first place, I would like to thank my supervisor Prof. Dr. Peter H. Schmitt for providing an excellent working climate and environment, fruitful discussions, steady support and help. In particular for his patience in the last months of writing the thesis and his steady encouragements.

I am grateful to Prof. Dr. Ulrich Furbach that he agreed to act as second reviewer for my thesis.

I am in deep debt of gratitude to my colleagues Dr. Andreas Roth and Dr. Steffen Schlager including their families for discussions, suggestion and their friendship.

Also I want to thank Prof. Dr. R. Hähnle for his support and to allow me to finish writing my thesis, while I should have worked on the MOBIUS project.

My thanks go also to the other co-funders of the KeY-project Prof. Dr. Bernhard Beckert and Prof. Dr. Wolfram Menzel for allowing me to participate in an exciting research project.

I want also to thank Prof. Dr. Wolfgang Ahrendt, Dr. Thomas Baar, Dr. Martin Giese, Vladimir Klebanov, Dr. Wojciech Mostowski, Philipp Rümmer, Angela Wallenburg and all other colleagues and also the many students of the KeY-project, it was always a great pleasure to work with you.

I want also to thank my colleague Frank Werner, who started in Karlsruhe a few month ago for the nice time and moral support.

Last, but not least my thanks go to my parents–Eva and Jörg Bubel–without them nothing would have been possible.

Gothenburg, May 2007
Richard Bubel

Deutsche Zusammenfassung

Die vorliegende Arbeit ist im Bereich der Programmverifikation angesiedelt und führt Methoden zur Verifikation verzeigerter Datenstrukturen ein. Diese Methoden wurden von mir im Rahmen meiner Arbeit für das KeY-Projekt entwickelt.

Das KeY-Projekt erforscht Fragestellungen, die sich im Rahmen der deduktiven Verifikation objekt-orientierter Programme ergeben. Insbesondere wurde und wird ein Verifikationswerkzeug entwickelt mit dem sich JAVA CARD Programme verifizieren lassen. Die zur Zeit umfassendste Beschreibung des KeY-Projekts findet sich in [BHS06].

In Kap. 2 der Arbeit werden die Grundlagen der im KeY-Projekt verwendeten dynamischen Logik JAVA CARD so weit wie für das Verständnis der Arbeit notwendig eingeführt. In weiten Teilen folgen die Definition denen in Kap. 3 im KeY-Buch [BHS06] angegebenen.

Im Mittelpunkt dieser Dissertation steht die Spezifikation und Verifikation von Eigenschaften, deren Definitionen in der Regel rekursiven Charakter haben. Dieser Art von Definitionen ist typischerweise bei der Spezifikation und Verifikation von Eigenschaften induktiv-definierter Datenstrukturen wie Listen oder Bäumen bzw. beliebiger Graphen anzutreffen.

Um Eigenschaften solcher Strukturen auf einfache und für die spätere Verifikation geeignete Weise ausdrücken zu können, wird in Kap. 3 eine neue Kategorie von zustandsabhängigen Symbolen eingeführt, die als expliziten Teil ihrer Syntax Informationen über die Zustandsabhängigkeit enthält. Aus dieser Information kann man schließen, ob eine Zustandsveränderung Einfluss auf den Wert eines solchen Symbols hat. Das Verwenden dieser hier erstmals eingeführten Symbolklasse führt zu einer signifikante Verbesserung der Effizienz des Kalküls. Im angeführten Kapitel wird zudem eine Auswahl von Anwendungsgebieten dieser Symbolklasse vorgestellt, wie die formale Definition von Erreichbarkeit in verzeigerten Strukturen oder die Representation sog. Queries, d.h. von Methoden deren Implementierung keine Veränderung des Speichers (Zustands) verursacht. Aufbauend auf der Erreichbarkeitsformalisierung werden systematisch Prädikate für die Spezifikation ausgewählter Datenstrukturen eingeführt.

Während sich das vorhergehende Kapitel auf Spezifikationsaspekte konzentriert, wird in Kap. 4 eine Regel eingeführt, die die Verifikation rekursiv implementierter Methoden in JAVA CARD DL ermöglicht. Bisher war eine Verifikation dieser Methoden in JAVA CARD DL nur möglich, wenn die maximale Rekursionstiefe durch eine feste und parameterunabhängige Zahl gegeben war. Die Anwendung der Regel wird anhand zweier Beispiele, wie die Implementierung einer Listenumkehr, vorgeführt.

Kap. 5 präsentiert eine allgemeine Vorgehensweise mit der sich verzeigerte Datenstrukturen unter Verwendung abstrakter Datentypen spezifizieren lassen. Der Vorteil ist, dass letztere von vielen implementationsabhängigen Details ab-

strahieren können und sich somit Beweise vereinfachen lassen. Ein ausgearbeitetes Beispiel wird in Kap. 6 vorgestellt, welches eine Formalisierung des JAVA-Datentyps `String` vorstellt. Nach bestem Wissen des Autors wurde eine solche Formalisierung in der Literatur bisher noch nicht beschrieben.

In Kap. 7 werden einige der vorgestellten Techniken im Rahmen einer Fallstudie zur Verifikation der Korrektheit einer Implementierung des Schorr-Waite Algorithmus vorgestellt. Der Graphenmarkierungsalgorithmus Schorr-Waite kommt im Gegensatz zu anderen Algorithmen seiner Art mit einem festen, nicht von der Größe des Graphen abhängenden, Platzbedarf aus. In Erweiterung zu anderen bisher durchgeführten Verifikationen des Algorithmus wird hier die Variante für beliebig verzweigende Graphen als korrekt nachgewiesen.

1 Introduction

1.1 The KeY Approach

The KeY-approach aims at the integration of deductive verification within the development process. Therefore a tight integration into standard development environments as CASE (computer aided design environment) tools and IDE (integrated development environments) is indispensable.

KeY adds specification and verification support to these tools. A semi-automatic prover builds up the system's core component that allows to prove that the implementation meets its specification, visualise possible thrown and uncaught exceptions or generate test cases. The taken approach allows to draw benefit even from incomplete specifications or not completed proofs.

As target programming language JAVA CARD has been chosen, which is mainly a subset of JAVA. At the moment of writing[1] this means no multi-threading, floating point operations or graphical user interface classes are supported including some other minor simplifications. But in addition to standard JAVA it comes with built-in support for transactions. KeY supports the *complete* JAVA CARD language and also most of the features previously summarised under *minor simplifications* like full object and class initialisation.

The supported high-level specification languages are

- the Unified Modelling Language (UML) in combination with the Object Constraint Language which is part of the UML specification [Obj01].
- the Java Modelling Language [LPC+02, LBR00] a specification language designed for the specification of JAVA programs.

these specifications become then translated into a variant of dynamic logic called JAVA CARD DL, which can also be seen as a low-level specification language. Using dynamic logic grants access to a wide repository of fundamental knowledge and experience concerning a sound theoretic underpinning, calculus design and others.

The thesis concentrates on handling non-rigid functions (resp. predicates), i.e., symbols whose interpretation depends on the (program) state. Several problems can be naturally specified on an acceptable abstraction level with the help of auxiliary non-rigid functions or predicate symbols.

We will demonstrate how they can be used to specify interesting aspects of inductively defined data structures like graphs, lists, etc. In these cases the non-rigid symbols are often defined in a recursive manner. Another well-known application area are (recursive) methods, which are in principle (recursively)

[1] the upcoming JAVA CARD specification will for example include multi-threading

defined non-rigid functions. In this thesis techniques are developed that allow an efficient formal treatment in proofs.

1.2 Structure of the Thesis

The thesis is structured as follows: Chapter 2 fixes the notation and definitions used throughout the thesis. It provides the theoretical underpinning of JAVA CARD DL–the base logic of the KeY-project–and follows in most parts the definitions given in Chapter 3 of the KeY-Book [BHS06].

The thesis is mainly concerned with the specification and verification of properties, whose definition is inherently recursive. The most common application scenarios for such properties originate from the need to specify or verify inductively defined data structures like lists, trees or, even more general, arbitrary graph data structures.

In order to specify such properties in a convenient way, a new class of state dependable (non-rigid) symbols is introduced in Chapter 3. In contrast to classical non-rigid symbols, they carry syntactic information allowing to deduce which state changes may or may not effect their value. This kind of dependence information allows a considerable improvement in the efficiency of the calculus when treating non-rigid symbols. Examples demonstrating their use for the formalisation of reachability properties and for the specification of inductively defined structures back up this statement. As a further application field, it is shown how the modelling of queries benefits from the use of this new kind of symbols.

While the former chapter is more tailored to specification issues, Chapter 4 presents a rule that allows to treat and verify recursive implemented methods within the JAVA CARD DL framework. Up-to-now recursive methods could only be treated if the maximal recursion depth had been a fixed value, which must not depend on any kind of parameters. The section gives two examples demonstrating the rule, such as a recursively implemented list reversal.

In Chapter 5 a general framework is presented that allows to connect a linked data structure to an abstraction defined in terms of an abstract data type. The advantage is that the specification of the linked data structure may abstract from details on the implementation level and concentrate on functional properties. An elaborated example is given in Chapter 6 specifying JAVA's String data type, which has to the best of the author's knowledge not been formally specified before.

The case study described in Chapter 7 demonstrates the use of some of the techniques introduced in this thesis. Its content is the specification and verification of Schorr and Waite's graph marking algorithm. The contained chapter is an extended version of the corresponding chapter in the KeY-Book, which has been written by the author of the thesis. The Schorr-Waite algorithm is used as a benchmark for program verification environments. The specified and verified JAVA-implementation is–again to the best of the author's knowledge–also the first one which had been done for an arbitrary (but finite branching) graph data structure.

Chapter 8 rounds off the work by summarising the presented work and pointing out open points and directions for further future work.

1.3 Related Work

Treatment of linked data structures is necessary within any program verification environment. The LOOP [vdBJ00] project used higher-order logics and a deep embedding of the JAVA semantics to provide a program verification system for JAVA. The use of higher-order logics and deep-embedding allows for a direct formalisation of reachability. As the method invocation stack is directly represented standard induction techniques could be used to treat recursive methods. Both approaches are not feasible within JAVA CARD DL. The disadvantage of deep embeddings is the high encoding overhead, which lowers–together with the expressivity of higher-order logics–the degree of automation and makes interaction difficult.

Another project exploiting higher-order logics is BALI [Ohe01], which is implemented in Isabelle/HOL and uses also a deep embedding. The advantages and disadvantages are comparable to the ones mentioned for LOOP.

The KIV [BRS+00] group in Augsburg added also support for JAVA to their verification system. KIV uses also a dynamic logic. In contrast to KeY, they model the heap explicitly as a logic data type, which allows e.g. for explicit quantification of the existence of a path between two objects. In JAVA CARD DL the heap is not explicitly modelled, only the heap changes are bookkept using so called updates. As updates are logic operators similar to syntactic substitutions and not represented as a logic data type this kind of quantification is not possible within KeY and other techniques needed to be developed.

Further there are several decision procedures for subclasses of graph structures implemented by SAT resp. SMT solvers like haRVey [ARR03]. Typical problems solved by those theories is if two given structures (in the supported class) are equal.

Related to heap structures is also an approach used in program analysis called *Shape Analysis*. Shape analysis maps concrete heaps to abstract ones by accumulating heap cell (nodes of graph) that cannot be distinguished by given so called instrumentation predicates resp. local variables into a summary node. The program is then executed on the abstract heap by abstract interpretation. A nice approach is presented in [SRW99] using three valued logic to model the abstract heap.

Part I

Foundations

2 The JAVA CARD Dynamic Logic

In order to reason about programs one needs a language that allows to make them a subject of discussion. A number of different languages has been and is used for this purpose. For example, programs can be represented as terms of a higher order logics together with an axiomatisation of the program language semantics. For the Java programming language, this approach has been taken by the LOOP [vdBJ00] and BALI [Ohe01] project using PVS [ORS92] resp. Isabelle/HOL [Pau94]. The advantages of using a higher order logic is a gain in expressiveness which also allows to reflect about properties of the programming language itself rather than about pure programs. Further, together with a theorem prover these fundamental approaches allow to prove mechanically consistency properties of the program language formalisation itself and so to gain some substantial confidence about its accordance. Furthermore, any calculus built on top of such a formalisation is to be guaranteed correct. Of course, also these approaches do not close the gap between an informal (though precise) language specification in a natural language and its translation into a strict mathematical formalism. The drawbacks of this approach are the encoding of programs in terms of the logic, which makes them hard to read and to orientate if a proof needs interaction. If one works directly on the program semantics axiomatisation one has to deal with a lot of internal details often not important for pure reasoning about programs. Building a good calculus on top of the axiomatisation that provides a reasonable abstraction is an additional major work.

Another well known language and more tailored to the purpose of program verification is the Hoare language/calculus. The main notion is that of a Hoare triple $\{P\}\alpha\{Q\}$ where P, Q are formulas in a first order logic and α an arbitrary program. The semantics of this triple is that in a state where P holds and when α terminates then after the execution of α the formula Q holds. In contrast to the former approach programs are not translated into logic terms and easy to recognise and read. Hoare triples are (mostly) limited to partial correctness and lack of closure concerning quantifiers. In order to overcome these drawbacks (and some more) KeY uses a different kind of logic that keeps programs as first citizen members (i.e., without translating them into terms) and which subsumes the Hoare logic. This logic called dynamic logic has been invented by David Harel [Har84]. Mainly in contains two new operators box $[\cdot]$ and diamond $\langle\cdot\rangle$ where $P \rightarrow [\alpha]Q$ has exactly the same meaning as the above Hoare triple. The formula that results from replacing the box by the diamond operator would in addition require program α to terminate in order to become true. In contrast to higher order logics one gets different instances of dynamic logics for different programming languages and usually for real-world languages the program semantics is specified in terms of a basic set of calculus rules. In

KeY we are convinced that the increase in readability and the good abstraction level from internal details are worth to use a logic tailored for program verification and therefore to put additional effort in approaches that allow to gain confidence in the correctness and consistency of the rules. This additional effort is done in KeY by providing some kind of reflection mechanism [BRR04] that allows to prove correctness of rules relative to a (small) set of axiom rules. A selected set of axiomatic rules have been crossverified against the formalisation of the Java program language semantics in Isabelle/HOL called BALI [Tre05]. The crossverification of the while invariant rule [Wid06] and the previously mentioned reflection mechanism have been co-supervised by the author of this thesis.

In the remaining chapter, the most important basic notions and concepts of the logic JAVA CARD DL are introduced. A full account of the used logic is given in [BHS06].

2.1 Syntax And Semantics

This section defines the logic JAVA CARD DL as far as necessary for this thesis. It is assumed that the reader is familiar with sorted first-order logics and classical first-order dynamic logic.

In the first part the signature, syntax and semantics of the logic is described. It is sketched how to set up a concrete instance of the logic for a given program environment. The definition of so called *updates* and the validity of formulas are of particular importance.

In the second part the general design of the calculus is described and discusses the logical axiomatisation of a few selected concepts of the JAVA programming language.

At the moment of writing the best source for a full account of the logic JAVA CARD DL can be found in [BHS06], which also served as basis for this section.

2.1.1 Type Hierarchy and Signature

As Java allows inheritance (for interfaces even multi-inheritance) the logic's sort system is modelled as a type hierarchy and allows multi-inheritance for any type.

Definition 1 (Type Hierarchy)**.** The *type hierarchy* to be used is modelled as a bounded lattice $(\mathcal{T}, \sqsubseteq)$ where

- the set $\mathcal{T}$ denotes a finite set of symbols whose elements are called *types*. It contains at least the *any* $\top$ and *bottom* $\bot$ type.
- the relation $\sqsubseteq$ defines a partial order on $\mathcal{T} \times \mathcal{T}$, which is a reflexive, antisymmetric and transitive relation. In order to form a *bounded* lattice there have to be an upper and lower bound. For these roles the designated sorts $\top, \bot$ are used:
  - for all $t \in \mathcal{T}$ the following holds: $(\bot, t) \in \sqsubseteq$ and $(t, \top) \in \sqsubseteq$

From now on, we will write $(a \sqsubseteq b)$ instead of $(a, b) \in \sqsubseteq$.

In order to model interfaces, abstract and normal classes precisely, the set of types $\mathcal{T}$ is disjointedly divided into

1. a set $\mathcal{T}_a$ of abstract types and
2. a set $\mathcal{T}_d$ of dynamic types

The idea is that a dynamic type has elements which are exactly of this type and not of a subtype. In contrast there is no element that of an abstract type A but not of a subtype of A. The *any* type is a dynamic type, whereas $\bot$ belongs to the abstract types.

The actual type hierarchy for a concrete instance of JAVA CARD $\mathrm{DL}_{(\mathcal{T},\sqsubseteq)}$ depends on the *program environment*. The program environment are all class declarations and interface declarations together with their inheritance structure. The type hierarchy is then constructed from the environment in a straight forward manner. That is to say, if there are two classes `A` and `B`, where class `B` extends class `A` then there are two logic types of the same name and with the same inheritance structure.

Any program environment includes at least all primitive types like **`boolean`** or **`int`** and all classes resp. interfaces mentioned in the Java Language Specification (inclusive all inherited types), like `java.lang.Object`, `java.lang.Throwable`, `java.lang.Exception` etc. as well as the `Null` type.

With the type hierarchy on hand, the predicate, function and variable symbols can be declared. In logics those declarations are traditionally collected at one place called *signature*, analogous to *C++* header files:

Definition 2 (Signature)**.** The *signature* $Sig_{(\mathcal{T},\sqsubseteq)}$ is defined relative to a given type hierarchy and consists of the pairwise disjoint sets

- PSym containing all predicate symbols,
- FSym containing all function symbols and
- VSym containing all variable symbols

and an *arity* function $\alpha : (\mathrm{PSym} \cup \mathrm{FSym} \cup \mathrm{VSym}) \to \mathcal{T}^+$ assigning each symbol its signature, i.e., argument types and result type. We define the arity n of a function symbol f as $n = |\alpha(f)| - 1$ and the one of a predicate symbol p as $n = |\alpha(p)|$, where $|\cdot| : \mathcal{T}^+ \to \mathbb{N}$ is the length(=number of types) of the word given as argument.

The predicate and function symbols are further partitioned into two disjoint classes, namely *rigid* and *non-rigid* symbols. While at this point this are only two notions, the interpretation of the non-rigid symbols will later on depend on the state of a program, while the symbols of the first will be state independent.

There are some predefined function and predicate symbols, which will be given a fixed interpretation when defining the semantics. The most important are:

- for any type T there is a rigid function $\sqsubseteq T : \top \to \texttt{boolean}$, intended to test if the given element is of type T (or one of its subtypes).
- for any type T (except $\bot$) there is a so called cast function (rigid) $(T) : \top \to T$ mapping any element to one of type T. The intention is that the cast function will be the identity on elements of type T (or one of its subtypes).
- the rigid constant **`null`** of type `Null`.
- the usual arithmetic operators (e.g., `+, -, *, /`) and comparators (e.g., $>=$, $<=$), where the first ones are modelled as rigid functions and the second ones as rigid predicates. Further there are rigid function symbols that allow to represent any natural number using the human readable decimal system.
- the boolean typed constants TRUE and FALSE.
- the non-rigid predicate symbol *inReachableState* explained in a later section

As for the type hierarchy a subset of the signature is derived from the program environment, which are modelled as *special* kinds of non-rigid function symbols:

- program variables and static attributes are modelled as non-rigid constants of the corresponding type.
- for any instance attribute `a` of type `T` declared in a class `C`, there is a unary non-rigid function symbol `a@(C)` : `C` $\to$ `T`.
- for any non-void method there is a non-rigid function symbols of the same signature.

Note 1. Program variables, (static) attributes and non-void method form own distinguishable subcategories of non-rigid functions.

The word subset is chosen with care as we further demand that there is an infinite number of function symbols for each arity.

2.1.2 Terms and Formulas in JAVA CARD DL

The definition of terms Trm_T for $T \in \mathcal{T}$, programs Prg and formulas Fml coincide widely with the standard definitions of sorted first-order logic resp. dynamic logic. The definitions are therefore kept short and concentrate on not so often found constructs:

Definition 3 (Terms)**.** The sets Trm_T for $T \in \mathcal{T}$ is defined as the smallest set satisfying

1. any variable symbol $v \in \mathrm{VSym}$ with $\alpha(v) = T$ is a term of type T and belongs therefore to Trm_T for $T \in \mathcal{T}$.

2. given a function symbol $f \in \text{FSym}$ with signature $\alpha(f) = T_1 \cdots T_n T$ and terms $t_1, \ldots, t_n$ with $t_i \in \text{Trm}_{S_i}$ where $S_i \sqsubseteq T_i$ then $f(t_1, \ldots, t_n)$ is in Trm_T for $T \in \mathcal{T}$.

3. given a formula (see Def. 4) $\phi \in \text{Fml}$ and terms $t_1 \in \text{Trm}_T$, $t_2 \in \text{Trm}_T$ then
$$\backslash\text{if } (\phi) \backslash\text{then } (t_1) \ \backslash\text{else } (t_2)$$
is in Trm_T for $T \in \mathcal{T}$.

4. given a variable symbol $v \in \text{VSym}$ of type S a formula (see Def. 4) $\phi \in \text{Fml}$, terms $t_1 \in \text{Trm}_T$, $t_2 \in \text{Trm}_T$ then
$$\backslash\text{ifEx } S\ v;\ (\phi) \backslash\text{then } (t_1) \ \backslash\text{else } (t_2)$$
is in Trm_T for $T \in \mathcal{T}$. The variable c is bound in ϕ and t_1, but not in t_2.

5. given an update u (see Def. 7) and a term $t \in \text{Trm}_T$ then $\{u\}\ t$ is a term in Trm_T.

Definition 4 (Formulas)**.** The set Fml of formulas is defined as the smallest set satisfying

1. given a predicate symbol p with signature $\alpha(p) = T_1 \cdots T_n$ and terms $t_i \in \text{Trm}_{S_i}$ where $S_i \sqsubseteq T_i,\ i \in \{1 \ldots n\}$ then $p(t_1, \ldots, t_n)$ is in Fml.

2. the definitions for composing formulas using negation !, conjunction &, disjunction |, implication -> or equivalence <-> are as usual.

3. given a variable symbol $v \in \text{VSym}$ of type T and a formula ϕ then $\backslash\text{forall } T\ v;\ \phi$ resp. $\backslash\text{exists } T\ v;\ \phi$ are formulas in Fml.

4. given formulas ϕ, ψ and $\tau \in \text{Fml}$ then
$$\backslash\text{if } (\phi) \backslash\text{then } (\psi) \ \backslash\text{else } (\tau)$$
is in Fml.

5. given a variable symbol $v \in \text{VSym}$ of type T and formulas ϕ, ψ and $\tau \in \text{Fml}$ then
$$\backslash\text{ifEx } S\ v;\ (\phi) \backslash\text{then } (\psi) \ \backslash\text{else } (\tau)$$
is in Fml. The variable v is bound in ϕ and ψ, but not in τ.

6. given an update u (see Def. 7) and a formula $\phi \in \text{Fml}$ then $\{u\}\ \phi$ is in Fml.

7. given a program $p \in \text{Prg}$(see Def. 5) and a formula ϕ then
   - $\langle p \rangle \phi$ (read: *diamond p phi*)
   - $[p]\phi$ (read: *box p phi*)

   are formulas in Fml.

Definition 5 (JAVA CARD DL Programs)**.** The set of all JAVA CARD DL programs Prg is defined as the smallest set containing all legal sequences of JAVA CARD DL statements.

Thereby a JAVA CARD DL statement is any JAVA statement as defined by the Java Language Specification [GJSB00] or a *method-frame* (see Def. 23) resp. *method-body* (Def. 22) statement.

A legal sequence is defined as a sequence of statements $st_1; \ldots; st_n;$, which when embedded in a static method of a public class in the default package would be a legal JAVA[1] program with respect to the program environment.

Note 2. In other words a sequence of statements $st_1; \ldots; st_n;$ is considered to be a JAVA CARD DL program, if

```
public class DefaultClass {

  public static void defaultMethod() {
    st_1;
    ...;
    st_n;
  }
}
```

is compiled by the JAVA compiler.

The last syntactical category to be defined here are *updates*. Updates are JAVA CARD DL specific, they have been introduced for the first time in [Bec01]. The most complete and formal treatment is given in [Rüm06].

Definition 6 (Syntactical Location)**.** A *syntactical location* is a non-rigid function representing

- a program variable pv, or
- a static attribute $T.a$ resp. an instance attribute $a@(T)$ declared in type T, or
- the array access operator []

of the corresponding arity.

Definition 7 (Update)**.** Given a syntactical location f and terms o_i and v. Then the expression $f(o_1, \ldots, o_n) := v$ denotes an *elementary update* (or assignment pair). Let now u, u_1, u_2 be updates then

- the sequential composition $u_1 \,; u_2$
- the parallel composition $u_1 \,||\, u_2$
- the conditioned $\texttt{\bs if}\ \phi;\ u$ where ϕ is a formula

[1] the notion legal for occurrences of method-frames or method-body statements will be given in Def. 23 resp. Def. 22

- and the quantified \for x; ϕ; u where x is a logic variable bound in formula ϕ and update u

are updates themselves.

Note 3. Updates assign locations (local program variables, fields, arrays, etc.) a fixed value. In fact they change the evaluation state of the term or formula occurring behind them. Instead of describing the complete state only the differences between the pre-state (state before the update) and the post state are enumerated.

Example 1. Some updates and their intended semantics:

- The elementary update $x := t$ assigns the program variable x the value t. Applying it to a program variable y, i.e., $\{x := t\}\ y$, is equal to t if x and y are the same program variables otherwise the term evaluates to y.
- The parallel update $x := y \,||\, y := x$ swaps the content of the program variables x and y. Please note that parallel updates are evaluated simultaneous, and therefore are independent of each other.
- The parallel update $x := 1 \,||\, x := 2$ is called inconsistent as it assigns the same location two different values (clash). In contrast to *abstract state machines (ASM)* KeY resolves these clashes. For parallel updates the clash semantics is a simple *last-one wins semantics*.
- The quantified update \for i; *true*; $a[i] := 0$ assigns all array components to the value 0. In case of a clash in a quantified update, e.g. \for i; *true*; $a[0] := i$ the clash semantics is that the assignment with smallest i satisfying the guard wins. Therefore the necessity of a well-order for the domain mentioned in Sec. 2.1.3. The latter quantified update assign a[0] the value 0 as the used well-order for integers is defined as $0, -1, 1, -2, 2, \ldots$.

Definition 8 (Anonymous Update)**.** The elements of the set $\mathit{AnonUpd} := \{*_1, *_2 \ldots\}$ are called *anonymous updates* and belong also to the category of updates.

Note 4. The idea behind anonymous updates is to erase any knowledge about the state a formula is evaluated to. In principal they behave like an anonymous program in classical dynamic logic, except that they always terminate.

2.1.3 Semantics of Java Card DL

As usual in modal and therewith dynamic logics the semantics of Java Card DL is defined in terms of a so called Kripke structure.

Definition 9 (Java Card DL Kripke Structure)**.** The structure $\mathcal{K} := (\mathcal{M}, \mathcal{S}, \rho)$ where

- $\mathcal{M} := (\mathcal{D}, I)$ is a partial first-order structure with domain $\mathcal{D}$ and an interpretation I for the rigid function and predicate symbols.

- a set of states $\mathcal{S}$ where a state is identified with the interpretation of non-rigid function and predicate symbols.
- the state transition relation $\rho : \text{Prg} \times \mathcal{S} \to \mathcal{S}$ where $\rho(p)(S)$ denotes that state reached when executing program p in state S. The transition relation is defined according to the semantics of JAVA as described in [GJSB00].

Note 5. The previous definition uses the notion partial in relation to the first-order structure $\mathcal{M}$. The reason is that the interpretation is – as mentioned – only defined for rigid function or predicate symbols. Those symbols are then interpreted as *total* functions or predicates, in particular there is no undefinedness value in JAVA CARD DL.

JAVA CARD DL models undefinedness by underspecification: Consider the following JAVA CARD DL term $div(\mathtt{1}, \mathtt{0})$ where $div : \mathbf{int} \to \mathbf{int}$ represents the arithmetic division. As JAVA CARD DL has an arithmetic structure those functions have a fixed interpretation. But in mathematics division by zero is not defined, therefore the interpretation I is only fixed for div, as long as the second argument does not evaluate to 0. Not fixed means that the value of the term $div(\mathtt{1}, \mathtt{0})$ may be evaluated differently in different JAVA CARD DL Kripke structures, but also that it is always evaluated to some value. A comparison and discussion of different approaches modelling undefinedness can be found in [HÖ5].

Note 6. The domain $\mathcal{D}$ is part of the partial first-order structure $\mathcal{M}$ and therewith the same in any state. Thus JAVA CARD DL follows a paradigm called *constant domain assumption*.

Let $\mathcal{K} := (\mathcal{M} := (\mathcal{D}, I), \mathcal{S}, \rho)$ denote a JAVA CARD DL Kripke structure. The domain $\mathcal{D} := (\mathsf{U}, \delta_0, \preceq)$ consists of

- the universe U, which is a non-empty set of elements. In the following the notation $e \in \mathcal{D}$ is often used, where in fact $e \in \mathsf{U}$ is meant.
- the function $\delta : \mathsf{U} \to \mathcal{T}$ assigns each element of the universe its dynamic type.
- JAVA CARD DL requires that a well-order $\preceq$ is defined on the universe U. The rationale of the well-order is shortly given later, but for the remaining thesis of no further importance.

Let the function $\delta^{-1} : \mathcal{T} \to \mathcal{P}(\mathsf{U})$ assign any type $T \in \mathcal{T}$ the smallest set of elements of U such that for any of its elements e the equation $\delta(e) = T$ holds. In particular $\delta^{-1}(T)$ is empty for all abstract types. Further it is required that for a non-abstract class type C the set $\delta^{-1}(C)$ contains an infinite number of elements.

The interpretation I assigns any type $T \in \mathcal{T}$ a subset of the universe such that

- $I(T) = \{\}$ if and only if $T = \bot$

- $I(\mathbf{int}) = \mathbb{Z}$ is interpreted as the whole numbers,
- $I(boolean) = \{tt, ff\}$,
- $I(\texttt{Null}) = \{null\}$,
- for any dynamic type T representing a class or interface (reference) type $I(T)$ is defined as $I(T) := \delta^{-1}(T) \cup \bigcup_{S \sqsubseteq T} I(S)$. The recursion is well-defined, because the type hierarchy is finite (or at least noetherian).

Note 7. Any reference type contains at they have at least the dynamic type `Null` as a subtype.

As the definitions for the semantics of terms and formulas does not differ significantly from the standard definition, they are only sketched in the succeeding. Terms and formulas are evaluated in a Kripke JAVA CARD DL structure $\mathcal{K} := (\mathcal{M}, \mathcal{S}, \rho)$, a state S and a variable assignment $\beta : \text{VSym} \to \mathcal{D}$ by the continuation of the interpretation I resp. S.

- Let $f(t_1, \ldots, t_n) \in \text{Trm}_T$ then $f(t_1, \ldots, t_n)^{(\mathcal{K},S,\beta)}$ is defined as
  - $I(f)(t_1^{(\mathcal{K},S,\beta)}, \ldots, t_n^{(\mathcal{K},S,\beta)})$, if f is a rigid function symbol.
  - $S(f)(t_1^{(\mathcal{K},S,\beta)}, \ldots, t_n^{(\mathcal{K},S,\beta)})$, if f is a non-rigid function symbol.
- Let $p(t_1, \ldots, t_n) \in \text{Fml}$ then $p(t_1, \ldots, t_n)^{(\mathcal{K},S,\beta)}$ is defined as
  - $I(p)(t_1^{(\mathcal{K},S,\beta)}, \ldots, t_n^{(\mathcal{K},S,\beta)})$, if p is declared rigid.
  - $S(p)(t_1^{(\mathcal{K},S,\beta)}, \ldots, t_n^{(\mathcal{K},S,\beta)})$, if p is declared non-rigid.

  Predicate symbols are mapped to the truth values $true, false$.
- $(\backslash\text{ifEx } T\ x;\ (\phi)\ \backslash\text{then}(t_1)\ \backslash\text{else}(t_2))^{(\mathcal{K},S,\beta)}$ evaluates to
  - $t_1^{(\mathcal{K},S,\beta')}$, if if there is an element e such that $(\backslash\text{exists } T\ x;\ \phi)^{(\mathcal{K},S,\beta)}$ evaluates to $true$. The variable assignment β' evaluates on x to the minimal element $min_{\preceq}$ (wrt. the well-order $\preceq$) satisfying the formula ϕ. For all other variable the assignment β' is identical with β.
  - $t_2^{(\mathcal{K},S,\beta)}$, otherwise. Note, that the variable assignment has not changed for this case.

The next four definitions define the semantic of updates and their application on terms or formulas.

Definition 10 (Semantic Location)**.** A *semantic location* is defined as a tuple $\langle f, (e_1, \ldots, e_n)\rangle$ where $f : T_1 \times \ldots \times T_n \to T$ is a syntactical location as defined in Def. 6 and $e_i, i \in \{1, \ldots, n\}$ are elements in $I(T_i) \subset \mathcal{D}$.

Definition 11 (Semantic Update)**.** An *elementary semantic update* is a pair $(\langle f, (e_1, \ldots, e_n)\rangle, d)$ where $f : T_1 \times \ldots \times T_n \to T$, $e_i, i \in \{1, \ldots, n\}$ are elements in $I(T_i) \subset \mathcal{D}$ and d an element in the domain belonging to type T. A possible empty set of elementary semantic updates is called *semantic update*.

Definition 12 (Consistent Semantic Update)**.** A semantic update is called *consistent*, if it contains for any semantic location at most one *elementary semantic update*.

Definition 13 (Application of a Consistent Semantic Update)**.** The application of a consistent semantic updates CU is a mapping between states. Applying CU on a state S maps S to a state $CU(S)$, which is identical with S except for the semantic locations occurring as part of the CU's semantic updates $(\langle f, (e_1, \ldots, e_n)\rangle, d)$ for which $S'(f)(e_1, \ldots, e_n)$ evaluates to d.

Definition 14 (Semantics of Anonymous Updates)**.** A JAVA CARD DL Kripke Structure $\mathcal{K}$ interprets an anonymous update $*_n$ as a consistent semantic update CU_n mapping all locations to fixed values, i.e., $CU_n(S) = T_n$ for all states $S \in \mathcal{S}$ to a state $T_n \in \mathcal{S}$.

Note 8. The interpretation of an anonymous update does not depend on the current state, but may vary between different JAVA CARD DL Kripke structures.

Note 9. The given definition of an anonymous update differs from the presentation given in [BHS06], where it is introduced as an abbreviation for a finite parallel update and constructed in relation to a given proof situation, i.e., sequent.

Definition 15 (Evaluation of a Syntactical Update)**.** Given a JAVA CARD DL Kripke structure $\mathcal{K}$, a state $S \in \mathcal{S}$ and a variable assignment β. A syntactical update u is evaluate to a consistent update as described below:

- the elementary update $u := f(t_1, \ldots, t_n) := v$ then $u^{(\mathcal{K},S,\beta)}$ evaluates to the semantic update $\{(\langle f, (t_1^{(\mathcal{K},S,\beta)}, \ldots, t_n^{(\mathcal{K},S,\beta)})\rangle, v^{(\mathcal{K},S,\beta)})\}$.
- the conditioned update $\backslash\texttt{if}\ \ \phi$; u is evaluated to the same consistent update as u, if $\phi^{(\mathcal{K},S,\beta)}$ is satisfied. Otherwise it is evaluated to the empty consistent update.
- the quantified update $\backslash\texttt{for}\ x$; ϕ; u is evaluated to

$$\overbrace{\{u^{(\mathcal{K},S,\beta^d_x)} \mid \text{f.a. } d \in \mathcal{D}\}}^{:=\mathcal{QU}} - \{\bar{u} \mid \text{f.a. } \bar{u} := (\langle f, (d_1, \ldots, d_n)\rangle, \bar{e}) \in \mathcal{QU} \text{ for which a } u := (\langle f, (d_1, \ldots, d_n)\rangle, e) \in \mathcal{QU} \text{ exists such that } e \prec \bar{e}\}$$

- the parallel update $u_1 \,||\, u_2$ is evaluated to the semantic update $CU_2 \cup C\bar{U}_1$, where CU_2 is the (consistent) semantic update $u_2^{(\mathcal{K},S,\beta)}$ and $C\bar{U}_1 := \overbrace{u_1^{(\mathcal{K},S,\beta)}}^{:=CU_1} - \{u := (\langle f, (d_1, \ldots, d_n)\rangle, d) \in CU_1 \mid \text{exists } u' := (\langle f, (d_1, \ldots, d_n)\rangle, e) \in CU_2\}$
  (abbr. $CU_1 \,||\, CU_2 := CU_2 \cup C\bar{U}_1$)
- the sequential update $u_1\,;u_2$ is evaluated to $u_1^{(\mathcal{K},S,\beta)} \,||\, u_2^{(\mathcal{K},u_1^{(\mathcal{K},S,\beta)}(S),\beta)}$

$$\text{andLeft}\ \frac{\Gamma \Longrightarrow \phi, \Delta \quad \Gamma \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \phi\ \&\ \psi, \Delta} \qquad \text{andRight}\ \frac{\Gamma, \phi, \psi \Longrightarrow \Delta}{\Gamma, \phi\ \&\ \psi \Longrightarrow \Delta}$$

$$\text{allLeft}\ \frac{\Gamma, \backslash\text{forall}\ \ x;\ \phi, \{\backslash\text{subst}\ x;\ t\}\phi \Longrightarrow \Delta}{\Gamma, \backslash\text{forall}\ \ x;\ \phi \Longrightarrow \Delta}$$

where t is an arbitrary ground term

Figure 2.1: Selected rules for the sorted first-order fragment in text book style

Lemma 1. *The evaluation of a syntactical update results always in a consistent semantic update.*

Proof. Sketch. Structural induction about the structure of syntactical updates. The only interesting cases are the ones for quantified and parallel updates. But their definition already takes care of the removal of duplicates of semantic locations. Together with the induction hypothesis that the composites u resp. u_1, u_2 are evaluated to consistent semantic updates, the lemma's proposition follows. □

Definition 16 (Application of Updates)**.** Let $\mathcal{K}$ denote a JAVA CARD DL Kripke structure, $S \in \mathcal{S}$ be a state and β a variable assignment. Given an update u, a term t resp. formula ϕ then an application of u on t or ϕ, i.e., $\{u\}\ t$ resp. $\{u\}\ t$ is evaluated as

$$t^{(\mathcal{K}, u^{(\mathcal{K},S,\beta)}(S),\beta)} \text{ resp. } \phi^{(\mathcal{K}, u^{(\mathcal{K},S,\beta)}(S),\beta)}$$

Definition 17 (Validity of Formulas)**.** Let $\mathcal{K}$ denote a JAVA CARD DL Kripke structure. A formula ϕ is satisfiable in $\mathcal{K}$ if there is a state $s \in \mathcal{S}$ and variable assignment β such that $\phi^{(\mathcal{K},s,\beta)}$ evaluates to true.

A formula *phi* is called valid if there is a JAVA CARD DL Kripke structure $\mathcal{K}$ such that $phi^{(\mathcal{K},s,\beta)}$ holds for all states $s \in \mathcal{S}$ and all variable assignments β.

A formula *phi* is called logically valid if ϕ is valid for all JAVA CARD DL Kripke structures $\mathcal{K}$.

2.2 Calculus

In KeY a sequent calculus is used to prove the logical validity of formulas. The rules for the classical first-order fragment of the logic are mainly identical to the rules found in any traditional formalisation of a sequent calculus.

For the design of the calculus special care has been taken to maintain well-typedness of terms and formulas when rules are applied. Consequently one had to restrict the application of rules applying equalities and to compensate these restriction by the introduction of type casts. The first-order fragment has been proven sound and complete (see [Gie05]) for the sorted first-order fragment of JAVA CARD DL (excluding arithmetic). An excerpt of these rules is shown in Fig. 2.1.

2.2.1 Symbolical Execution

Symbolic execution, i.e., one allows symbolic values as input for programs, is the basic paradigm followed by most of the calculus rules treating programs. This means a complex statement is stepwise decomposed into a sequence of simpler statements until an elementary (not further decomposable) statement is reached. For example:

JAVA
```
  x=a[++i] + 3;

becomes to

  int j = i + 1;
  int y = a[j];
  x = y + 3;
```
JAVA

One problem to treat is how to decompose two statements, in classical dynamic logic one simply replaces the diamond containing the program by a nested one, where the first diamond simply contains the first statement of the program and the second one the rest:

```
\<{ st1;st2;st3; }\> phi ---> \<{ st1; }\>\<{ st2;st3; }\> phi
```

In JAVA such a rule cannot be (easily) provided due to nested try-catch blocks, method-frame blocks, or just simple labelled and unlabelled blocks, e.g.

```
\<{ label:{
     try {
       st1;
       st2;
     } catch(Exception e) { st3; }
 }\> phi
---> ???
```

The solution taken by KeY is the introduction of the notion of the *first active statement*. This means the rules dealing with programs focus on the first statement occurring after a prefix of opening blocks and rewrite this statement stepwise into a sequence of simpler ones until an elementary statement is reached. An elementary statement is then, for example an assignment where the left and right hand side have no side-effects.

2.2.2 Rules and Taclets

Fig. 2.1 lists already some rule of a first-order calculus in textbook style. These rules are represented in KeY using taclets. The taclet language is an easy to learn language in which logical rules can be expressed. This section will give only some examples necessary to get an intuitive understanding, taclets are described in detail in Chapter 4 of [BHS06] and in [BGH+04, Gie03].

The following taclet renders the andRight rules:

KeY

```
andRight {
    \schemaVar \formula b,c;
    \find (==> b & c)
    \replacewith(==> b);
    \replacewith(==> c)
    \heuristics(split,beta)
};
```

KeY

Any taclet has a unique name, here andRight. In addition, they can have an additional displayname (not shown here) which is only presented to the user and needs not to be unique. The first line defines the schema variables `b` and `c` of type formula, i.e., they match only on formulas and not on (logic) terms.

The find section of the taclet identifies the pattern to look for in the sequent, in the above case a top level conjunction in the sequents succedent. In the succeeding lines a semicolon separated list of goal templates is given describing the resulting goals of a taclet application. A goal template is a white space separated list of at most one replacewith and at most one add statement. As expected a replacewith causes the term focused on in the find part to be replaced, while the add statement simply adds a formula to the antecedent resp. succedent of the sequent.

The heuristics section gives hints for the automatic application, but has no logic meaning.

Another example of a taclet is the applyEq rule:

KeY

```
applyEq {
  \schemaVar \term G s;
  \schemaVar \term H t;
  \assumes (s = t ==>)
  \find (s)
  \sameUpdateLevel
  \replacewith ( t )
  \heuristics ( apply_equations )
   \displayname "apply␣equality"
};
```

KeY

In contrast to the former rule, the given rule is a pure rewrite rule which matches on a term `s` and replaces the occurrence by a term `t`, but only when the equation `s = t` is available in the antecedent. The last side condition is expressed by the assumes section, its logical meaning is equivalent to an additional goal template adding the equation to the succedent. This means if the user applies this taclet and the equation is not syntactically present in the sequent, then an additional goal opens where one has to prove that the equation holds. Also new is the `\sameUpdateLevel` flag, which allows an application of the taclet only if the occurrence of the focused term `s` is in the same state as

the occurrence of the assumes clause, e.g. the rule would not be applicable in a case like:

```
o.a = 0 ==>  {o.a:=3} o.a = 0
```

As a last example for a taclet, may serve the following program transformation taclet treating a post increment expression:

KeY

```
postincrementAssignment  {
   \find (\<{.. #lhs0 = #lhs1++; ...}\> post)
   \varcond (\new(#v, \typeof(#lhs0)))
   \replacewith (
     \<{..
         #typeof(#lhs0) #v = #lhs1;
         #lhs1 = (#typeof(#lhs1))(#lhs1+1);
         #lhs0 = #v;
       ...}\> post)
   \heuristics(simplify_int)};
```

KeY

The find section matches on a postincrement assignment expression when it is the first active statement, i.e., only preceeded by a sequence of opening braces, try-statements, labels or methodFrame headers. The latter mentioned sequence is matched by `..`, while the rest of the program following after the postincremet is matched by `....` The expression .. · ... is also called *program context schemavariable*.

The above rule is also an example for the use of variable condition, which can add further side conditions on the instantiations of schemavariables, in this case that te introduced program variable must be not used yet in the proof.

2.2.3 Object Creation

KeY supports object creation and initialisation as specified in paragraph 12.5 of the Java Language Specification (JLS). A full account of object creation and initialisation can be found in [Bub01, BP06, BHS06]. This section describes only the formalisation of the object creation aspect, i.e., the allocation of a new object.

The dynamic logic follows the constant universe paradigm, i.e., all states share the same domain. The formula \forall T o; $\phi(o)$ quantifies about *all* – even not yet created – objects. Consequently the creation of a new instance cannot be modelled by adding new elements to the universe. Rather object creation is seen and treated as a change of the objects respective classes state. In order to describe the formalisation in more detail some new symbols need to be introduced.

Definition 18 (Object Repository)**.** In the context of object creation we use the term *object repository* of a non-abstract class type T for the set of elements with dynamic type T. The object repository of a type T is always interpreted as an infinite and enumerable set.

In order to achieve a canonical representation of elements of an object repository of a type T and to access its elements we define a access function:

Definition 19 (Object Repository Access Function)**.** For any non-abstract class type T there exists a rigid function symbol T::get : $int \to T$. The function symbol is interpreted as an injective function whose image is equal to the object repository of T.

With the above definition the objects o and u of dynamic type T with $o \doteq$ T::get(i) and $u \doteq$ T::get(j) are equal if and only if $i \doteq j$ is valid.

Definition 20 (Object index)**.** The *index of an object* o is the integer i such that $o \doteq$ T::get(i) is valid.

The above definitions give us a syntactical representation for any object in the universe. On top of this we can define the notion of a created object. As a preliminary step we require any non-abstract class type to declare a private static integer field `<nextToCreate>`. These pre-existent, non-user declared fields are called *implicit fields*.

Definition 21 (Created Object)**.** In a state S, we call an object of dynamic type T *created*, if its index is non-negative and less than the value of the static implicit field T.`<nextToCreate>` is evaluated to in state S.

For convenience reasons there is an implicit Boolean field `<created>` declared in class `java.lang.Object`, which has to be set true if an object is created.

When a new object of dynamic type T is created (allocated) by a Java program, i.e., an instance creation expression in Java is transformed to a call of an implicit static method `<allocate>()` declared by any non-abstract class type. The return value of the method is then specified to be the object with index T.`<nextToCreate>`. Simultaneously the value of T.`<nextToCreate>` is incremented by one and the implicit field `<created>` of the object is set to true. The taclet specifying the allocation method is:

KeY

```
instanceAllocationContract {
  \find (\modality{#allmodal}
     {.. #t(#lhs)::#t.#allocate(); ...}\endmodality(post))
  \varcond(\hasSort(#t, G))
  \replacewith ({#lhs := G::<get>(#t.<nextToCreate>) ||
      #t.<nextToCreate> := #t.<nextToCreate> + 1 ||
      G::<get>(#t.<nextToCreate>).<created> := TRUE}
      \modality{#allmodal}{..  ...}\endmodality(post))
   };
```

KeY

Note 10. For incrementing the value of `<nextToCreate>` the mathematical addition is used and must be used. Otherwise, an overflow would occur when the maximal value of Java integers is reached.

2.2.4 Java Reachable States

The Kripke structure of our logic contains several states that can never be reached by a Java program. Besides others this is a consequence of using implicit static or instance fields, that can be changed by an update, to encode the state of an object, e.g., if it is created. As described in Sect. 2.2.3 the used implicit fields are intended to be non-negative (e.g., `<nextToCreate>`) respective to be consistent with the objects state like `<created>`. The problem is that both properties cannot be ensured syntactically. The user can easily destroy these properties by entering an update that sets the fields to inconsistent values, e.g., when applying the `cut`-rule.

The problem can be and is solved by introducing the non-rigid predicate *inReachableState*, which is satisfied in a state S, if and only if, state S can be reached via a JAVA program. In the calculus the predicate is axiomatised by formulas, which express the above mentioned consistency properties.

For example, if we are in a state where the predicate is known to be true, we can safely use the attribute `<created>` to express or demand that an object has to be created or to reason that objects referenced from created objects are created or `null`. There are special suited rules, which allow to exploit this kind of properties and other rules that allow to prove that an update leads to a JAVAReachable State.

For the second part the naive approach by just expanding the predicate to its axioms would lead to inefficiently long and deep formulas. Therefore the rule expanding the predicate, analysis preceeding updates and generates a stronger, but noticeable shorter proof obligation. The author of the thesis has been deeply involved in formalising the optimised rules.

2.2.5 Symbolical Execution of Method Invocations

Treatment of method-calls is always a challenging aspect of program analysis and verification, as it comes with a change of context and the kind of solution decides how far the used method scales up on larger systems.

The solution sketched in this section follows the symbolic execution paradigm, but has substantial drawbacks with respect to modularity. The modular treatment of method calls is described in Sect. 2.2.6.

In JAVA CARD a method is invoked when evaluating a method reference statement. The typical shape of an instance method reference statement is `r=o.m(e1,..,en);`, where `o` is an expression evaluating to an object called receiver of the method invocation, `r` denotes a location used to keep the return value of the method invocation and the expressions `e1` to `en`, whose values are handed over as arguments. Details like static or void methods are left out here, as their treatment is nearly equal–but simpler[2]–to the one of value typed instance methods.

In a first phase the method reference statement is evaluated from the left to the right in order to determine the location where to store the return value, the receiver instance and the argument values to be handed over. If during these

[2]At least as long as ignoring static initialisation, which is not subject of this thesis.

```
1 B resLoc = u;
2 if (resLoc == null) {
3    throw new NullPointerException();
4 }
5 A receiver = o;
6 E1 v1 = e1;
7 ...
8 En vn = en;
9 R j = receiver.m(v1, ..., vn); resLoc.result = j;
```

Figure 2.2: Sequence of statements resulting from the symbolic execution of the method reference statement `u.result=o.m(e1,..,en);`

steps an exception or error is raised the evaluation of the method reference statement stops and the exception is propagated through.

In the context of symbolic execution this means a statement like the one above with

- expression `o` being of type `A`,
- the location `r` keeping the return value denoting an attribute location of type `R`, e.g. `u.result` where `u` is an expression of type `B`, and
- the expressions `e1` to `en` being of type `E1` to `En`

is transformed to the sequence of statement shown in Fig. 2.2.

In the following it is assumed that the symbolical execution has reached line 9. At this point the calculus has to dynamically dispatch the actual used method. Therefore the calculus splits the proof into two further branches. The first one deals with the case that the `receiver` pointed to **`null`** in which case a `NullPointerException` is thrown. The second proof branch assumes that the `receiver` is not **`null`** and relies solely on the dynamic type of `receiver` and the arguments static types for method dispatching. An **`if`** statement cascade is created in order to test the type of `receiver` via **`instanceof`** from the bottom most to the upper most type of classes implementing a method with a matching signature. Assume three classes `A2` extending `A1` extending `A` and let method `m` be declared abstract in class `A` and implemented in both subclasses. The **`if`** cascade then looks like:

```
if (self instanceof A2)
   j=receiver.m(v1,...,vn)@(A2);
else
   j=receiver.m(v1,...,vn)@(A1);
```

The statement of the `then`- resp. **`else`**-branch is called *method body statement* and, merely, a placeholder for the method body itself.

Definition 22 (Method Body Statement)**.** A statement matching the following syntax

$$(resultVar=)_{opt} receiver.methodName(argVar_1, \ldots, argVar_n)@className)$$

where

- $resultVar$ is a variable of type T_0,
- $receiver$ is a variable of type T,
- $argVar_1, \ldots, argVar_n$ are variables of types $T_1, \ldots, T_n$, and where,
- $methodName : T_1, \ldots, T_n$ is the signature of a method [3] actually implemented by class $className$ being a subtype of type T

is called a *method body statement*.

Execution of a method body statement means to

1. assign each parameter variable the value of the corresponding argument (variable). The introduction of fresh parameter variables ensures also that assignments to those variables are only local, i.e., do not influence the value of the variables handed over as arguments.
2. insert the method body of the referred method implementation embraced by a so called *method frame*. A method frame can be seen as the symbolic representation of an element of the method stack keeping track of the current program context in which the statements are executed. Program context means the instance referred to by a **this** reference and the class context in which the statements are executed. The latter one is crucial for visibility issues.

The syntax of a method frame is given in the following definition:

Definition 23 (Method Frame)**.** The *method frame* statement consists of a header storing the program context information and a program block with statements executed in the context described by the header. The syntax is:

```
method-frame(result -> variable,
             this   =  variable,
             source =  className):{
  ... // statements
}
```

Note 11. If inside the method frame a **return** statement is executed, the value to be returned is assigned to the variable given in the result section of the method-frame header.

Assuming that in the previous example the dynamic type of `receiver` has been `A2` the symbolical execution leads to the following piece of program:

```
method-frame(result->j,this=receiver, source=A2):{
  T1 p1 = v1;
  ...
  Tn pn = vn;
  ... // method body
}
```

[3] In JAVA the return type does not belong to the signature of a method.

2.2.6 The Method Contract Rule

Symbolical execution of a method invocation has some serious drawbacks. For any invocation one has to repeat most of the proof steps, which have been already done somewhere else. In case of overwritten methods, it is in general necessary to split the proof into several branches – one for each implementation. It is obvious that this kind of method treatment leads to huge and redundant proof trees. Another disadvantage is that proofs with an unfolded method implementation tend to be not modular. This manifests itself when changing or overwriting a method implementation forces all proofs, which expanded that particular method, to be redone. More details about modular verification can be found in [Rot06].

In order to circumvent or at least reduce these problems one aims at an atomic treatment of method invocations. In other words, the calculus switches temporarily from a more small step to a big step semantics. Therefore one encodes the visible behaviour of the method into a logic formula. A method invocation is then replaced by the formula describing the post-state of the method.

Usually this formula is not generated, but provided by the developer in terms of a method contract. If an implementation of a method complies to the specified contract, the contract can be used instead of the method implementation. The compliance of a method's implementation to its contract has to be proven. Normally, this proof is the only one that requires to unfold the method and consequently, the only proof that has to be redone in case of a contract compliant change of the method implementation.

A method contract $C := (\mathcal{P}, mbs, pre, post, mod)$ consists of

- a set $\mathcal{P}$ of program variable declarations, which can be used in all constituents of the contract.
- the method body statement *mbs* serves to identify the exact method being the subject of the contract. The statement has the following shape

  `result=self.m@(par1,...,parN)@(T)`

  with the obvious alterations in case of static method invocations or void methods. The symbols `result`, `self` and all method parameters are program variables declared in $\mathcal{P}$.
- the formula *pre* called precondition that has to be satisfied in order to benefit from the contract.
- the formula *post* called postcondition, which describes the effects of the method, or in other words, which is ensured to be satisfied when the control flow returns from the method invocation.
- a set *mod* has to contain all locations that may have been changed at the end of a method execution.

In a more general setting a method contract contains also a termination marker that indicates whether termination is guaranteed or not. All contracts that will be used or treated throughout this thesis require normal termination.

Therefore the syntactical add-ons needed to specify the behaviour in presence of control flow redirections due to uncaught exceptions are skipped here.

Proving compliance of a method specification and its implementation. An implementation of a method is called compliant to a method contract C, if it can be proven that in any state, for any method call receiver and any combination of parameters satisfying the precondition, the postcondition holds after the method execution returns.

For a fixed contract C a formula po_C can be generated which formalises the above description:

$$\forall selfLV : T; \forall \overrightarrow{parLV}; \{\texttt{self} := selfLV \,||\, \overrightarrow{\texttt{pars}} := \overrightarrow{parLV}\}$$
$$(pre \texttt{ -> } ((\langle \texttt{result = self.m(par1,...,parN)@(T);} \rangle post) \;\&\; po_C^{mod}))$$

The generated formula is called proof obligation of the method contract C. If the formula is proven valid one has shown that for any receiver object and any combination of method argument values that satisfy the method's precondition *pre*, the method will terminate and in its final state its postcondition *post* holds. It should be noted that even if not explained further the modifies clause of the method needs to be proven correct. How to formalise the modifies clause proof obligation po_C^{mod} is, for example, described in [Rot06].

Using method contract C. The rule, which replaces a method invocation using its contract C can be written in textbook style as:

$\mathsf{useMethodContract_C}$

$$\frac{\Gamma \Longrightarrow \{self := o \,||\, \overrightarrow{par} := \overrightarrow{args}\}\ (pre \;\&\; \mathcal{V}_{mod}(post \texttt{ -> } \langle \texttt{..} \quad \texttt{...} \rangle \phi))}{\Gamma \Longrightarrow \langle \texttt{..} \quad \texttt{result=o.m(args)@(T); ...} \rangle \phi}$$

The modifies set *mod* is used to generate an anonymising update $\mathcal{V}_{mod}$, which sets all locations contained in *mod* to a fixed, but unknown value by assigning them the value of a new introduced constant or, more general, function. If the modifies set contains all available locations the anonymous update(see Def. 8) is taken for $\mathcal{V}_{mod}$.

The method contract rule for a method contract C is not sound on its own behalf. But soundness can be ensured when the proof obligation formula po_C can be shown valid.

Theorem 1. *The* $\mathsf{useMethodContract_C}$ *rule is sound, if the proof obligation formula* po_C *can be proven valid without using the method contract rule itself.*

Allowing `Let` definitions in contracts. `Let` definitions are a common construct in many formal languages. Writing specifications and using them later becomes significantly simpler when providing at least some basic support. The syntax of a `Let` expression is

$$\backslash\texttt{letFunc}\ T \quad f(T_1\ p_1, \ldots, T_n\ p_2) \;= fctDefTrm\ \backslash\texttt{in}$$

or

$$\texttt{\textbackslash letVar}\ T\ x;\quad \texttt{\textbackslash suchThat}\ \Phi(x)\ \texttt{\textbackslash in}$$

Formally, `Let` definitions introduce new *contract local* rigid function symbols resp. logic variables. Each `Let` function symbol declares a sequence of logical variables as part of its signature. These variables are restricted to occurrences in the function's definition term (here:$fctDefTrm$).

The (possible) values of the `Let` variables are characterised by a specifier provided formula. All those variables are universally bound on the complete contract. There is also a strong relation ship between this kind of variables and *epsilon*-terms as, for example, treated in [Gie98].

The function defining term resp. the variable characterising formula is evaluated in the pre-state of a method invocation. All function declarations must occur in front of the variable declarations. The textual order of the definitions plays also a role for visibility issues of the symbols in other `Let` definitions of the contract as only previously declared symbols are visible in another `Let` definition. Note, that in particular no `Let` declared variable can occur as part of the definition of a `Let` declared function. All `Let` functions or variables are accessible in the pre- and postcondition as well as in the modifies set. The justification for these restrictions is to prevent inconsistencies that could occur in presence of recursive definitions resp. to keep the method contract rule understandable and usable.

A method contract C^{let} with support for `Let` definitions is also referred to as $C^{let} := (pv, (letFunc)*, (letVar)*, pre, post, mod)$.

In postconditions one is often interested in the value of a term or the definition of a non-rigid function (e.g. an attribute) as it has been in the method's pre-state. The `Let` definitions supply both needs.

Example 2. The class `PayCard` models a paycard with an integer typed field `balance` that keeps the currently loaded amount of money. The method `charge` loads a specified amount of money onto the paycard.

```
PayCard self; int amount;

\method: self.transferMoney(amount)@(PayCard)

\letFunc int balance@pre(PayCard p1) = p1.balance \in
\pre : amount >= 0
\post: self.balance = balance@pre(self) + amount
\modifies : self.balance
```

The proof obligation formula $po_{C^{let}}$ for a contract with `Let` definitions is

defined as:

$$
\begin{array}{l}
\forall selfLV : T_{self};\ \forall \overrightarrow{parsLV} : \overrightarrow{T_{mbs}};\ \{\texttt{self} := selfLV \,||\, \overrightarrow{\texttt{pars}} := parsLV\}\ (\\
\text{// for any } \texttt{Let} \text{ function declaration } f_i(\overrightarrow{par_{f_i}}) \\
\quad (\quad \underbrace{(\forall \overrightarrow{pars_{f_1}} : \overrightarrow{T_{pars_{f_1}}};\ f_1(\overrightarrow{pars_{f_1}}) = fctDefTrm_{f_1})}_{:= trans_{f_1}} \\
\qquad \& \ldots \& \\
\qquad \underbrace{(\forall \overrightarrow{pars_{f_r}} : \overrightarrow{T_{pars_{f_r}}};\ f_r(\overrightarrow{pars_{f_r}}) = fctDefTrm_{f_r})}_{:= trans_{f_r}} \\
\quad) \text{ -> } \\
\qquad \forall \overrightarrow{letVars} : \overrightarrow{T_{letVars}};\ \big((\phi_1(letVar_1) \,\&\, \ldots \,\&\, \phi_s(letVar_s)) \\
\qquad\quad \text{ -> } \\
\qquad\qquad (pre \text{ -> } \langle \texttt{self.m(par1,...,parN)@(T);} \rangle post) \\
\qquad\qquad \& \\
\qquad\qquad (pre \text{ -> } po^{mod}_{C_{let}}) \\
\qquad\quad) \\
)
\end{array}
$$

The proof obligation quantifies first over all possible receivers of a method call and the whole range of parameter combinations. Most of the shown proof obligation is identical to the one for contracts without a `Let` expression. Thus only some words on the translation of

`Let` defined functions: Each `Let` expression is translated into a logic equation

$$\texttt{\textbackslash letFunc}\ T\ f(pars_f)\ = fctDefTrm_f\ \texttt{\textbackslash in} \rightsquigarrow f(pars_f) \doteq fctDefTrm_f$$

of which is then universally closed by all-quantification about the parameter variables. The function symbol itself is put to the signature of the logic. All such translations are connected into a conjunction, which then implies the remaining contract.

`Let` defined variables: For `Let` declared variables the translation is a bit less straight forward. In a first step one concatenates all variable characterising formulas to a conjunction. The conjunction implies then the remaining proof obligation, which is identical to the inner part of the proof obligation for contracts without `Let`s. The so constructed formula is then universally closed.

The $\mathsf{useMethodContract}_{\mathsf{C^{let}}}$ rule is now

$$
\frac{\begin{array}{l}
\{self := o \,||\, \overrightarrow{par} := \overrightarrow{args}\} \\
\quad (trans_{f_1} \,\&\, \ldots \,\&\, trans_{f_r} \\
\Gamma \Longrightarrow \quad\quad \& \ (\exists \overrightarrow{letVars} : \overrightarrow{T_{letVars}};\ (\\
\qquad\qquad \phi(letVar_1) \,\&\, \ldots \,\&\, \phi(letVar_s) \\
\qquad\quad \& \ pre \ \& \ \mathcal{V}_{mod}(post \text{ -> } \langle \ldots \quad \ldots \rangle \phi))))
\end{array}}{\Gamma \Longrightarrow \langle \ldots \quad \texttt{result=o.m(args)@(T);} \ \ldots \rangle \phi}\ \mathsf{useMethodContract}_{\mathsf{C^{let}}}
$$

Note 12. The `Let` variables, which are bound universally in the proof obligation formula, have to be bound existentially when using the contract.

Part II

Structural Specification and Verification

3 Structural Specification with Recursive Predicates

3.1 Location Dependent Non-Rigid Symbols

3.1.1 Motivation

Defining auxiliary function and predicate symbols is indispensable for specification purposes. Not only as abbreviations for complex terms and formulas, but also to allow the specification of recursively defined functions or predicates for which no closed expression exists. Even in the presence of an arithmetic structure the need to introduce new symbols will arise sooner or later.

The introduction of auxiliary *non-rigid* symbols leads to some general problems independent whether their definition is axiomatised recursively or given as closed expression, i.e., as term or formula without a self-reference.

The following list shall give only an impression of typical required non-rigid auxiliary symbols:

1. In order to specify reachability between two objects o_1, o_2 the introduction of a non-rigid recursive defined predicate `reach` is useful. The predicate is then specified such that `reach(`o_1`,`o_2`)` holds iff. there is a chain of given attributes connecting object o_1 with o_2 on the heap.

2. A useful abbreviation is the predicate `nonNull` defined on reference arrays, such that `nonNull(`a`)` is satisfied, if no component of the array `a` is equal to `null`.

Obviously the above listed predicates do not only depend on the values of their arguments, but also on the state in which they are evaluated: The reachable predicate depends not only on the value of the concrete objects o_1 and o_2, but also on the interpretation of the non-rigid attribute functions that make the connection between both objects. A similar situation holds for the `nonNull` predicate, whose value does not only depend on the array object a given as argument, but also on the values of a's components.

The problem with these implicit state dependencies can be easily illustrated using the following small example

Example 3. Let the program variable `a` denote an array of element type `java.lang.Object` and the program variable `i` be of type **`int`**. The sequent

$$\{\texttt{i} := 4\}\ \texttt{nonNull(a)} \Longrightarrow \texttt{nonNull(a)}$$

illustrates the practical problems when proving with general non-rigid predicates. The sequent cannot be closed easily, although obviously valid as the value of program variable `i` has no influence on the truth value of `nonNull`.

The problem is that the update simplification rules have no information about the locations on which the truth value of an auxiliary predicate depends. In order to prove the sequent of the example valid, one either has to insert the definition of the predicate (here: \forall *int* i; (`a[`i`] !` $\doteq$ `null`)) or to write and add special update simplification rules for any new introduced non-rigid symbol.

The first solution renders the use of auxiliary predicates nearly useless as it forces the insertion of the definition in too many cases, which in case of recursively defined predicates results in lengthy subproofs (requiring induction) or may even not work at all.

The second solution is not so good since it would require to extend the calculus for each introduced auxiliary predicate and to prove these additional rules sound. Beside the amount of work, as auxiliary predicates are often used by specifiers one cannot expect that they know how to extend the calculus. These considerations allow to reject this solution.

This chapter introduces therefore a new variant of non-rigid predicates, which carry explicit syntactic information about the locations on which their value depends on. In order to justify the introduction of this new class of symbols, two applications are described.

3.1.2 Syntax and Semantics

The principle idea is to encode the set of locations on which the value of a non-rigid symbol depends in their syntax and thus to make those dependencies explicit. To represent this set of locations an already existing formalism is reused. JAVA CARD DL provides already a flexible formalism to describe a set of locations in order to describe the modifies set of a method. The syntax of this formalism is derived from the syntax of quantified updates:

Definition 24 (Location Descriptor)**.** The syntax of a *location descriptor* is defined as

$$\texttt{\textbackslash for } \texttt{x}_1, \ldots, \texttt{x}_\texttt{n}; \texttt{\textbackslash if}(\phi)\ \texttt{loc}$$

where

- $x_1, \ldots, x_n$ are variables to be bound in ϕ and `loc`,
- ϕ is an arbitrary formula and
- the term `loc` must have a location as top level operator, i.e., a program variable, an attribute function or the array access function.

Example 4. Some examples of location descriptors

- `\for List x; \if(true) x.next@(List)` describes the set of all `next` locations of type `List`.
- `\for int i; \if(!(i` $\doteq$ `5)) a[i]` contains all – but the fifth – component locations of the array `a`

Definition 25 (Extension of a Location Descriptor)**.** Let $\mathcal{K} := (\mathcal{M}, \mathcal{S}, \rho)$ be a JAVA CARD DL Kripke structure with universe $\mathcal{D}$. The *extension of a location descriptor* ld for a given state $S \in \mathcal{S}$ is the set of locations, the location descriptor ld evaluates to in $(\mathcal{K}, S)$:

$$\underbrace{(\backslash \texttt{for } \texttt{x}_1, \ldots, \texttt{x}_n; \backslash \texttt{if}(\phi)\ \texttt{l}(\texttt{t}_1, \ldots, \texttt{t}_n))}_{=:ld}{}^{(\mathcal{K},s)} =$$
$$\{\langle\, l,\ (t_1, \ldots, t_n)^{(\mathcal{K},S),\beta}\rangle \mid \beta \models \phi, \beta : \mathrm{VSym} \to \mathcal{D}\}$$

The extension of a list or set of location descriptors is the union of their extension sets.

Attaching location descriptors to non-rigid function and predicate symbols makes the locations on which the value of such a symbol depends, explicit:

Definition 26 (Signature and Syntax: Location Dependent Symbols)**.** Let $ld;$ denote a semicolon separated list of location descriptors. The non-rigid predicate symbol

$$p[ld;] : T_1 \times \cdots \times T_n$$

is called *location dependent predicate symbol*. Analogously defined are *location dependent function symbols*.

The definition of terms and formulas remains unchanged. The only difference is that the signature contains the above defined location dependent symbols.

There are only few restrictions on the interpretation of ordinary non-rigid function and predicate symbols, mainly that the interpretation has to be well-defined and to respect the types. The situation is different for the interpretation of location dependent symbols. The interpretation of a location dependent symbol $p[ld;]$ has to coincide on states S_1, S_2 in which

1. the location descriptor extension of $ld;$ is the same, and
2. for any location $\langle f, (d_1, \ldots, d_n)\rangle$ in the extension set of $ld;$ the value of the $f^{(\mathcal{K},S_1)}(d_1, \ldots, d_n)$ and $f^{(\mathcal{K},S_2)}(d_1, \ldots, d_n)$ is equal.

Definition 27 (The relation $\approx_{ld;}$)**.** Let $\mathcal{K} := (\mathcal{M}, \mathcal{S}, \rho)$ denote a JAVA CARD DL Kripke structure, and $ld;$ a semicolon separated list of location descriptors.

For two states $S_1, S_2 \in \mathcal{S}$, the relation $S_1 \approx_{ld;} S_2$ holds iff. the location descriptor set extension of $ld;$ is equal in both states and

$$f^{(\mathcal{K},S_1)}(t_1^{(\mathcal{K},S_1)}, \ldots, t_n^{(\mathcal{K},S_1)}) = f^{(\mathcal{K},S_2)}(t_1^{(\mathcal{K},S_2)}, \ldots, t_n^{(\mathcal{K},S_2)})$$

holds for any tuple $\langle f, (t_1^{(\mathcal{K},S_1)}, \ldots, t_n^{(\mathcal{K},S_1)})\rangle \in ld;^{(\mathcal{K},S_1)}$.

Lemma 2. *The relation $\approx_{ld;}$ is an equivalence relation.*

Proof. Follows directly from the definition. □

Note 13. The above definition iterates only over the locations described by one location descriptor (here: $ld;^{(\mathcal{K},S_1)}$) in order to test if the values stored at the locations are the same. This is sufficient as the first part of the definition already requires that the extension sets are equal in both states.

Definition 28 (Semantics)**.** Let $\mathcal{K} := (\mathcal{M}, \mathcal{S}, \rho)$ denote a JAVA CARD DL Kripke structure, and ld; a semicolon separated list of location descriptors.

For any two states $S_1, S_2 \in \mathcal{S}$, the atomic formula $p[ld;](t_1, \ldots, t_n)$ evaluates to the same truth value, if $S_1 \approx_{ld;} S_2$ holds. Analogously for location dependent function symbols.

Example 5. Earlier in this section a normal non-rigid predicate `nonNull` has been used to abbreviate the property that all component values of a given array are not **`null`**. Now the location dependent predicate symbol

```
nonNull[\for Object[] o;\for int i; o[i];]
```

can be used for the same purpose. The reader should note that this formalisation over-approximates the set of locations the predicate depends on.

3.1.3 Update Simplification

With knowledge of the locations on which the value of a predicate or function symbol depends, the update simplification rules become more fine grained and it is possible to drop certain assignment pairs (see Def. 7) rather then accumulating all of them in front of any non-rigid symbol. We present a rule which allows to determine precisely whether an elementary update can be dropped in front of an update or if it has to be maintained. The rule implemented in the KeY system is a safe approximation of this rule. The presented rule may be available for interactive application in a future version of KeY.

In order to decide if an elementary update can be dropped in front of a location dependent function or predicate symbol, i.e., if

$$\{\underbrace{l := v}_{\mathcal{U}}\}\ p[ld;](s_1, \ldots, s_n) \rightsquigarrow p[ld;](s_1, \ldots, s_n)$$

the following conditions must be satisfied: Let S_{pre} denote the pre-state of $\mathcal{U}$ and S_{post} the state into which the update $\mathcal{U}$ leads to

1. the extensions $ld^{S_{pre}}$ and $ld^{S_{post}}$ are equal,
2. the updated location l is not contained in $ld^{S_{pre}}$ and
3. the arguments $s_1, \ldots, s_n$ evaluate to the same value in S_{pre} and S_{post}

The above properties can be expressed as logical formulae which have to be proved valid. Let $ld_i := \texttt{\textbackslash for } \overrightarrow{x_i}; \texttt{\textbackslash if}(\phi_i)\ \texttt{f}(\texttt{t}_1, \ldots, \texttt{t}_n)$ denote a location descriptor where $\overrightarrow{x_i} := \langle x_{i1} \ldots, x_{im} \rangle$.

In order to prove that the extensions of location descriptors are the same in S_{pre} and S_{post} the following formulae must be satisfied.

$$\forall\ \overrightarrow{x_i}.(\phi_i \rightarrow \exists\ \overrightarrow{x_i'}.(\{\mathcal{U}\}\ \phi_i' \wedge t_1 \doteq \{\mathcal{U}\}\ t_1' \wedge \ldots \wedge t_n \doteq \{\mathcal{U}\}\ t_n')) \tag{3.1}$$

$$\forall\ \overrightarrow{x_i'}.(\{\mathcal{U}\}\ \phi_i' \rightarrow \exists\ \overrightarrow{x_i}.(\phi_i \wedge t_1 \doteq \{\mathcal{U}\}\ t_1' \wedge \ldots \wedge t_n \doteq \{\mathcal{U}\}\ t_n')) \tag{3.2}$$

where $\phi_i' := \phi_i\{x_1/x_1', \ldots, x_n/x_n'\}$ where x_i' are new not already used variables of the same sort as their counterpart x_i (analogously t_i' are obtained from t_i).

Formalising property 2 can be achieved as follows: If the top level operator of l is not f, i.e., the same location operator as in the location descriptor ld_i, then l cannot be in its extension. Otherwise, if $l = f(r_1, \ldots, r_n)$

$$\forall \overrightarrow{x_i}.(\phi \rightarrow \neg(t_1 \doteq r_1) \vee \ldots \vee \neg(t_n \doteq r_n)) \tag{3.3}$$

where $freeVar(r_i) \cap \{x_{i1}, \ldots, x_{im}\} = \emptyset$

Checking if property 3 is met is a merely standard task, it has only to be shown that

$$(\{\mathcal{U}\}\ s_i) \doteq s_i. \tag{3.4}$$

Theorem 2 (Soundness)**.** *Let $\mathcal{K}$ denote a JAVA CARD DL Kripke structure, S an arbitrary state and $\mathcal{U}$ an elementary update $l := v$. If the formulae 3.1-3.2 are valid in $(\mathcal{K}, S)$ then*

$$(\mathcal{K}, S) \models \{\mathcal{U}\}\ p[ld;](s_1, \ldots, s_n) \leftrightarrow p[ld;](s_1, \ldots, s_n)$$

Proof. Let S' denote the state the update $\mathcal{U}$ leads to starting from state S, i.e., $S' = S_l^v$. Without loss of generality, let the list of location descriptors contain exact one descriptor: $ld := \backslash\texttt{for}\ \overrightarrow{x}; \backslash\texttt{if}(\phi)\ \texttt{f}(\texttt{t}_1, \ldots, \texttt{t}_{\alpha(\texttt{f})})$.

Assume S, S' coincide on the evaluation of the argument terms s_i, but they differ in the evaluation of $p[ld;](s_1, \ldots, s_n)$. From the definition of location dependent predicates we derive that the extension of the location descriptor depends on location l or that l is an element of the extension itself.

1. Again w.l.o.g., assume an $L := \langle f, (d_1, \ldots, d_{\alpha}(f))\rangle \in ld^{\mathcal{K},S}$, but $L \notin ld^{\mathcal{K},S'}$. According to Def. 25 there is a variable assignment β such that $\mathcal{K}, S, \beta \models \phi$, but for all variable assignments γ where $\mathcal{K}, S', \gamma \models \phi$ there exists an $i \in \alpha(f)$ such that $d_i \neq t_i^{\mathcal{K},S',\gamma}$. It is now obvious that in this case we fail to show that formula 3.1 holds.

2. For the case that l is part of the location descriptor's extension and had no influence on the extension itself, one can easily recognise that it is not possible to prove 3.4.

□

Despite the number and length of these formulas in most cases they collapse into trivial tasks. Nevertheless, the update simplifier in KeY uses a safe approximation of this rule and drops an elementary update only in the case that the updated location operator is not used at any place in the succeeding formula.

3.1.4 Soundness Proof Obligation for Axiomatisations of Location Dependent Symbols

Writing sound rules for location dependent symbols becomes slightly more difficult than for normal symbols. The reason is that one has to ensure that no

dependency on locations, which are not covered by the symbol's location descriptor, is introduced. In this section a proof obligation formula is defined excluding this additional source of error. If the proof obligation formula can be proven valid, it is ensured that a given axiomatisation of a location dependent symbol is sound (consistent) with respect to Def. 28.

From now on it is assumed w.l.o.g. that there is exactly one rewrite rule, which replaces a formula or term with a location dependent symbol as top level operator by its axiomatisation resp. definition. The axiomatisation (definition) may contain the symbol itself again (recursive definition).

Let $p[ld;]$ be a location dependent predicate with arity $\overrightarrow{T_{p[ld;]}}$. Let further be formula ϕ its axiomatisation, such that

$$\forall \overrightarrow{x} : \overrightarrow{T_{p[ld;]}};\ (p[ld;](\overrightarrow{x}) <\!\!-\!\!> \phi(\overrightarrow{x}))$$

holds. For the rewrite rule $\mathsf{axiom}_{\mathsf{p[ld;]}}$

$$p[ld;](t_1, \ldots, t_n) \rightsquigarrow \phi(t_1, \ldots, t_n)$$

where the t_is are terms of the appropriate type, the proof obligation $po_{p[ld;]}$ is defined as

$$\forall \overrightarrow{x} : \overrightarrow{T_{p[ld;]}};\ (\{*_n\}\ \{\mathcal{V}_{ld;}\}\ \phi(\overrightarrow{x})) <\!\!-\!\!> \{\mathcal{V}_{ld;}\}\ \phi(\overrightarrow{x}))) \tag{3.5}$$

where

- the update $\mathcal{V}_{ld;}$ is the anonymising update for the location descriptor $ld;$.
- the symbol $*_n$ belongs to the anonymous updates as introduced in Def. 8.

Theorem 3. *If the formula $po_{p[ld;]}$ is logically valid, then the rewrite rule $\mathsf{axiom}_{\mathsf{p[ld;]}}$ is correct with respect to Def. 28.*

Proof. In order to prove the soundness of rule $\mathsf{axiom}_{\mathsf{p[ld;]}}$ one has to show that:

If the formula $po_{p[ld;]}$ as defined above is logically valid, i.e., holds in all JAVA CARD DL Kripke structures **then**

for all JAVA CARD DL Kripke structures $\mathcal{K} := (\mathcal{M}, \mathcal{S}, \rho)$, for all states $S_1, S_2 \in \mathcal{S}$ and for all variable assignments β the following must hold:

If $S_1 \approx_{ld;} S_2$ (see Def. 27) then

$$\mathcal{K}, S_1, \beta \models \phi(x)$$
$$\textbf{iff.}$$
$$\mathcal{K}, S_2, \beta \models \phi(x)$$

The proof is performed indirectly. Given a JAVA CARD DL Kripke structure $\mathcal{K}'$, two states S_1', S_2' with $S_1' \approx_{ld;} S_2'$ and a variable assignment β', such that w.l.o.g. $\mathcal{K}', S_1', \beta' \models \phi(x)$, but $\mathcal{K}', S_2', \beta' \not\models \phi(x)$.

From these a witness Kripke structure $\hat{\mathcal{K}}$, state $\hat{S}$ and variable assignment $\hat{\beta}$ is constructed in which $po_{p[ld;]}$ is not satisfied. Consequently $po_{p[ld;]}$ is not logically valid.

Choose $\hat{\mathcal{K}}, \hat{S} \in \hat{\mathcal{S}}$ so that

1. the anonymising update $\mathcal{V}_{ld;}$ in 3.5 constructed from the location descriptor ld; evaluates to a consistent semantic update $CU_{\mathcal{V}_{ld;}}$ mapping state $\hat{S}$ to state S_1'.

2. the anonymous update $*_n$ maps any state to state S_2'.

It is always possible to select $\hat{\mathcal{K}}, \hat{S} \in \hat{\mathcal{S}}$ which fulfil both point and is equal to $\mathcal{K}'$ for all other symbols.

Rationale:

1. the anonymising update $\mathcal{V}_{ld;}$ assigns all locations fixed, but arbitrary values. Anonymising updates are created as quantified parallel updates assigning any location a rigid term with a not yet used constant or function symbol as top level symbol. *Not yet used* means, the symbols do not occur yet in the concrete proof. The proof obligation formula to be constructed and shown logically valid, is the only formula in the (root sequent of the) proof at construction time, this implies in particular that the top level constant and function symbols are not used anywhere else in $po_{p[ld;]}$.

   Consequently one can select a Kripke structure and therewith an interpretation of the rigid symbols such that $\mathcal{V}_{ld;}$ assigns all locations the value they have in S_1'. For $\hat{S}$ one can then select any state coinciding with S_1' on all other locations not covered by $ld;$.

2. anonymous updates are updates that evaluate to an infinite consistent update which assigns any possible location a fixed value. The interpretation of an anonymous update is state independent and depends like rigid symbols only on the chosen Kripke structure. In other words, any Kripke structure interprets an anonymous update $*_i$ as a function $*_i : \mathcal{S} \to \mathcal{S}$, which maps all states to exact one state $S_i \in \mathcal{S}$. For $\hat{\mathcal{K}}$ one selects from all Kripke structures satisfying point 1 the one interpreting $*_n$ as the function mapping all states to state S_2'.

For the evaluation of the left side of the equivalence in $po_{p[ld;]}$ one gets

$$(\{*_n\}\ \{\mathcal{V}_{ld;}\}\ \phi(\overrightarrow{x}))^{(\hat{\mathcal{K}},\hat{S},\hat{\beta})} \equiv (\{\mathcal{V}_{ld;}\}\ \phi(\overrightarrow{x})))^{(\hat{\mathcal{K}},S_2',\hat{\beta})} \equiv_{S_2' \approx_{ld;} S_1'} (\phi(\overrightarrow{x})))^{(\hat{\mathcal{K}},S_2',\hat{\beta})}$$

and the right side evaluates to

$$(\{\mathcal{V}_{ld;}\}\ \phi(\overrightarrow{x})))^{(\hat{\mathcal{K}},\hat{S},\hat{\beta})} \equiv (\phi(\overrightarrow{x})))^{(\hat{\mathcal{K}},S_1',\hat{\beta})}$$

which are not equivalent by assumption and thus $po_{ld;}$ cannot be proven logically valid. □

3.1.5 Modelling Queries

Definition 29 (Depends Clause of a Query)**.** A *depends clause* of a query is a list of location descriptors containing those locations on which the return value of the query depends. This means when the query is started in two states $S_1 \approx_{dep} S_2$, where *dep* is the depends clause of the query, then the queries termination behaviour and result must be the same.

A query with a depends clause can be now modelled using a location dependent rather than a general non-rigid function.

Definition 30 (Proof Obligation for the Depends Clause)**.** Given a query `q` of class `C` with depends clause $dep_{\texttt{q@C}}$. The proof obligation formula for the depends clause $dep_{\texttt{q@C}}$ is defined as:

$$
\begin{aligned}
&po_{dep_{\texttt{q@C}}} := \\
&\quad \forall \overrightarrow{resultLV} : T_{resultLV}; \forall \overrightarrow{argsLV} : \overrightarrow{T_{argsLV}}; \forall selfLV : \texttt{C}; \\
&\qquad \{\texttt{self} := selfLV \,||\, \overrightarrow{args} := \overrightarrow{argsLV}\} \\
&\qquad\quad ((\{*_1\}\ \{\mathcal{V}_{dep_{\texttt{q@C}}}\}\ \alpha_1)\ \&\ (\{*_2\}\ \{\mathcal{V}_{dep_{\texttt{q@C}}}\}\ \alpha_2) \texttt{ -> } resultLV_1 \doteq resultLV_2)
\end{aligned}
$$

with

- `self` is a program variable of type `C` and $\overrightarrow{args}$ a sequence of program variables with types corresponding to the queries signature.
- $\alpha_i := [\texttt{result}_i\texttt{=self.q(}\overrightarrow{args}\texttt{)@(C);}]resultLV_i = result_i$ and
- $\mathcal{V}_{dep_{\texttt{q@C}}}$ being the anonymising update constructed from *dep*.

Note 14. It is already assumed that the method is a query, i.e., its modifies set must be empty.

Theorem 4. *Given the proof obligation formula $po_{dep_{\texttt{q@C}}}$ for the implementation of query `q` in class `C`. If $po_{dep_{\texttt{q@C}}}$ can be proven valid, then $dep_{\texttt{q@C}}$ is a correct depends clause.*

3.1.6 Specification and Verification of a Sorting Algorithm

The verification of a sorting algorithm seems not to be an application area for location dependent functions, but it is. The here presented case study has been done together with Steffen Schlager, who was interested in an application example for the improved loop invariant rule he introduces in [Sch07].

The JAVA implementation of the sorting algorithm to be verified is shown in Fig. 3.1. In general the specification of a sorting algorithm requires to specify that afterwards

1. the array is sorted wrt. some given order.
2. the resulting array is a permutation of the original one.

It is the second item on the list, where location dependent symbols come into play. In order to express the permutation property, one introduces an auxiliary predicate

$$\texttt{perm[\textbackslash for(jint[] o; int i) \textbackslash if(0 <= i < o.length)o[i]](jint[], jint[]);}$$

such that `perm` holds in a state iff. the first array is a permutation of the second array. Furthermore, its location descriptor states that it only depends on array components and on array-length locations.

The proof obligation to show that the sorting algorithm is correct is given as shown below:

```
public void sort(int[] a) {
  int l=a.length;
  int pos=0;
  while (pos<l) {
    int counter = pos;
    int idx = pos;
    while (counter<l) {
      if (a[counter] > a[idx]) {
         idx=counter;
      }
      counter++;
    }
    int tmp  = a[idx]
    a[idx]   = a[pos];
    a[pos++] = tmp;
  }
}
```

Figure 3.1: Implementation of a selection sort algorithm for integer arrays

KeY

```
1 inReachableState ->
  \forall jint[] aCp; {\for int i; aCp[i]:=a[i]}
3   ((a!=null & a.length > 0 & a != aCp & aCp != null
      & perm[\for (jint[] o; int i)
5            \if (0<=i & i<o.length) o[i]](a, aCp))
     -> \<{ Sort.selectionSort(a)@Sort; }\> (
7           \forall jint i;(0 <= i & i < a.length-1 ->
                                        a[i] >= a[i + 1])
9           & perm[\for (jint[] o; int i)
                  \if (0<=i & i<o.length) o[i]](a, aCp)))
```

KeY

The precondition requires that

- array `a`, which will be sorted, is not **`null`** and contains at least one element.
- array `aCp`[1] intended to hold a copy of the original array must be different from `a`, i.e., not aliased and thus not affected when the component values of `a` might be changed.
- array `aCp` is a permutation of array `a` (line 4).

[1] allows to access the components of array `a` in the post state as JAVA CARD DL itself does not provide an `\old` resp. `@pre` operator as known from high-level specification languages like JML or OCL.

KeY

```
inReachableState  &
0 <= pos & pos <= a.length
& \forall jint x; (x >= 0 & x < pos-1 -> a[x] >= a[x+1])
& \forall jint x; (x>=0 & x <= pos-1 ->
     \forall jint y; (y >= pos & y < a.length -> a[x] >= a[y]))
& perm[\for (jint[] o; int i)
            \if (0 <= i & i < o.length) o[i]](a, aCp)
```

KeY

Figure 3.2: Invariant for the outer loop of the selection sort algorithm

Then one has to prove that the execution of method `selectionSort` terminates and that afterwards `a` has been sorted in descending order and that is still a permutation of `aCp` (line 9). Please note, that the component values of `aCp` have not been changed due to the not-aliased precondition.

During the proof one has to apply the loop invariant rule twice. Besides others, the used invariants have to state that the permutation is preserved all the time. The invariant for the outer loop is shown in Fig. 3.2.

For the proof of the selection sort algorithm, it is sufficient to exploit that a permutation remains a permutation, when swapping two elements. The rule allowing to prove this property is called swapPreservesPermutation and shown in Fig. 3.3. The correctness of this rule has to be (and has been) proven, therefore it has to be shown that it can be derived from the definition of the permutation predicate (see Fig. 3.3). Its definition uses itself a recursively defined location dependent function `mult`[\for(`jint`[] `o`; `int i`)`o`[`i`]]. The function `mult` counts the occurrences of a given element in a specified range of the array, e.g. `mult`[..](`a`, 4, 2, 5) counts the occurrences of 4 in the sequence of the values of `a[2]`, `a[3]` and `a[4]`.

3.2 Reachable Predicate

3.2.1 Syntax and Semantics

A further application of location dependent non-rigid symbols is the reachable predicate. An instance of a reachable predicate adheres to the following syntactical restriction:

Definition 31 (Reachable Predicate - Syntax)**.** The ternary location dependent non-rigid predicate `reach`[accessorList;](*T*,*T*,`int`) is called *reachable predicate*, where *accessorList* is a semicolon separated list of location descriptors obeying a restricted syntactic form, namely:

$$\underbrace{\texttt{\textbackslash for T x;\textbackslash if}(\phi)\ \texttt{x.attr}}_{Type\ \text{I}} \quad or \quad \underbrace{\texttt{\textbackslash for T x; int i;\textbackslash if}(\phi)\ \texttt{x.attr}[idx(\texttt{x},\texttt{i})]}_{Type\ \text{II}}$$

where *idx* is a function of integer type.

KeY

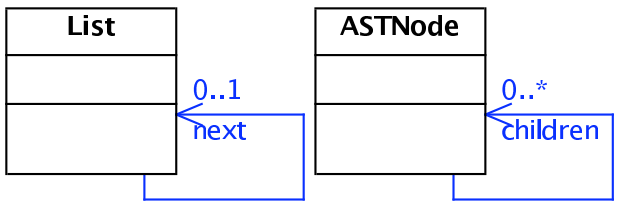
```
permDefinition {
    \find (perm[\for (o; i)
              \if (0<=i & i<o.length) o[i]](a,b))
    \varcond(\notFreeIn(n,a,b))
    \replacewith (\forall n;
      (mult[\for (o; i) o[i]] (a, n, 0, a.length) =
       mult[\for (o; i) o[i])](b, n, 0, b.length)))
};

multDefinition {
  \find ( mult[\for (o; i) o[i]](a, el, start, end) )
  \replacewith(
     \if (start >= 0 & start < end)
     \then
       ((\if (a[start] = el) \then (1) \else (0)) +
        mult[\for (o; i) o[i]](a, el, 1+start, end))
     \else (0))
};

swappingPreservesPermutation {
  \find (perm[\for (o; i)
              \if (0<=i & i<o.length) o[i])](a,b) ==>)
  \add(idx1 >=0 & idx2 >= 0 &
       idx1 < a.length & idx2 < a.length ->
         {a[idx1]:=a[idx2] || a[idx2]:=a[idx1]}
            perm[\for (o; i)
                 \if (0<=i & i<o.length) o[i]](a,b) ==>)
};
```

KeY

Figure 3.3: Permutation rules



(a) List (b) Graph

Figure 3.4: UML class diagrams modelling a list and graph

Example 6. Let `o`,`u` as well `s`, `t` be program variables of type `List` resp. `ASTNode` (see Fig. 3.4) and `n` be an arbitrary integer constant, then

- $\texttt{reach}[\backslash\texttt{for List x};\backslash\texttt{if(true) x.next}](\texttt{o},\texttt{u},\texttt{n})$ and
- $\texttt{reach}[\backslash\texttt{for ASTNode x};\texttt{int i};\backslash\texttt{if(i} >= 0 \wedge \texttt{i} < \texttt{x.children.length))}$ $\texttt{x.children[i]}](\texttt{s},\texttt{t},\texttt{n})$

are formulas used to specify that from list node `o` one can reach `u` via `next` attribute chains resp. that node `t` can be reached from node `s` via the children array attribute in *exactly* `n` steps

For the definition of the reachable predicate's semantics the following abbreviations will be useful: Let $accessorList := ld_1;\ldots;ld_k$ where each ld_i has one of the shapes required by Def. 31, then $l_j(x, idx_j(i,x)?)$ stands either for $x.attr$ if $ld_j := \backslash\texttt{for T x};\backslash\texttt{if}(\phi_\texttt{j})\ \texttt{x.attr}$ or $x.attr[idx(x,i)]$ if the j-th location descriptor is of type II.

The semantics of the reachable predicate needs to be defined. The definition will have to adhere the constraint given in Def. 28.

Definition 32 (Reachable Predicate - Semantics)**.** Let s, e denote terms of type T and n an integer term. Then $val_{(\mathcal{K},S,\beta)}(\texttt{reach}[accessorList;](\texttt{s},\texttt{e},\texttt{n}))$ evaluates to true iff $(\mathcal{K},S,\beta) \models$

$$n >= 0 \wedge \exists i_1,\ldots,i_\texttt{n};(l_{j_\texttt{n}}(\ldots \underbrace{l_{j_1}(\overbrace{\texttt{s}}^{s_0}, idx_{j_1}(\texttt{s},i_1)?)}_{s_1},\ldots,idx_{j_\texttt{n}}(s_\texttt{n},i_\texttt{n})?) \doteq \texttt{e} \wedge$$

$$\phi_{j_1}\{x/s_0, i/i_1\} \wedge \ldots \wedge \phi_{j_\texttt{n}}\{x/s_{\texttt{n}-1}, i/i_\texttt{n}\} \wedge \textstyle\bigwedge_{0<j<n} !\,end(s_j))$$

The formula *end* specifies that the access chain is not continued behind certain elements (for example, null).

The semantics definition is a bit cumbersome to read, due to the used unified notation for attribute and array access. The following two examples should clarify its meaning:

Example 7. The reachable predicate

$$\texttt{reach}[\backslash\texttt{for List x};\backslash\texttt{if(true) x.next}](\texttt{start},\texttt{end},\texttt{n})$$

of the previous `List` example is satisfied in $(\mathcal{K},S,\beta)$ iff.

$$(\mathcal{K},S,\beta) \models \texttt{start}\underbrace{\texttt{.next}\ldots.\texttt{next}}_{n} \doteq \texttt{end}$$

and

$$(\mathcal{K},S,\beta) \models !\,end(\texttt{start}\underbrace{\texttt{.next}\ldots.\texttt{next}}_{i})$$

for any $0 <= i < n$. A natural choice for the formula/predicate *end* would be $end(x) :\Leftrightarrow x \doteq \text{null}$.

Example 8. The meaning of the `ASTNode` reachable predicate in combination with $end(x) :\Leftrightarrow (x \doteq \text{null} \mid x\texttt{.children} \doteq \text{null})$ is then:

$(\mathcal{K}, S, \beta) \models$
$\quad \texttt{reach}[\backslash\texttt{for ASTNode x;int i;}\backslash\texttt{if}(inBound(\texttt{i}))\ \texttt{x.children[i]}](\texttt{s},\texttt{e},\texttt{n})$
iff.
$\backslash\text{exists } int\ k_1,\ldots,k_n;\ \backslash\text{exists ASTNode } l_1,\ldots,l_n;\ ($
$\quad l_1 \doteq \texttt{start.children}[k_1]\ \&\ \ldots\ \&\ l_n \doteq l_{n-1}\texttt{.children}[k_n]\ \&\ l_n \doteq \texttt{end}\ \&$
$\quad (0 <= k_1 < \texttt{start.children.length}\ \&\ \ldots\ \&$
$\qquad\qquad 0 <= k_n < l_{n-1}\texttt{.children.length})\ \&$
$\quad (\texttt{start}! \doteq \text{null}\ \&\ \texttt{start.children}! \doteq \text{null}$
$\quad\ \&\ \ldots\ \&$
$\quad\ l_{n-1}! \doteq \text{null})\ \&\ l_{n-1}\texttt{.children}! \doteq \text{null}))$

3.2.2 Calculus Rules for the Reachable Predicate

In order to use the reachable predicate in practise the above definitions need to be rendered as calculus rules. Therefore it is necessary to rewrite the above definition into an equivalent recursive version.

Let $accessors := ld_1;\ldots;ld_k$ be defined as before, where ld_j is either of type I or II. The reachable predicate $\texttt{reach}[accessors;](\texttt{s},\texttt{e},\texttt{n})$ can then be recursively defined as follows:

$\texttt{reach}[accessors;](\texttt{s},\texttt{e},\texttt{n}) :\Leftrightarrow$
$\quad (s \doteq e\ \&\ n \doteq 0)\ \mid$
$\qquad (n > 0\ \&\ !\,end(s)\ \&$
$\qquad\ \backslash\text{exists } T\ step;\ (\texttt{reach}[accessors;](\texttt{step},\texttt{e},\texttt{n}-1)\ \&$
$\qquad\ \bigvee_{j\in 1\ldots k}(recStep_j)))$
where $recStep_j :=$
$\quad s.acc_i \doteq step\ \&\ \phi_j\{x/step\}$, if ld_j is of type I
$\quad \backslash\text{exists } int\ m;\ (s.attr_j[m] \doteq step\ \&\ \phi_j\{x/step, i/m\})$, if ld_j is of type II

The big disjunction ranging from 1 to k enumerating all possible steps away from the concrete object is finite and fixed for an actual reachable predicate.

The above definition can be directly used as a rule that inserts the axiomatisation of a reachable predicate in a direct manner. The resulting taclet for the reachable predicate to express reachability for an `ASTNode` structure can the be given as follows:

— KeY —

```
reachableDefinition {
  \find(reach[...](s, e, n))
  \varcond(\notFreeIn(m, s, e, n),
           \notFreeIn(step, s, e, n))
  \replacewith (
     (s = e & n = 0) |
       (n > 0 &
         !(s = null | s.children@(ASTNode) = null) &
           n > 0 &
           \exists step;(reach[...](step, e, n-1) &
```

```
            (\exists m;( m >= 0 &
                         m < s.children@(ASTNode).length &
               (s.children@(ASTNode)[m] = step &
                reach[...](step, e, n-1)))
   ))))
};
```

KeY

The described rule schema defines the reachable predicate recursively, but the recursion is well-founded as for a negative path length n the formula evaluates to false and and each unwinding causes n be decremented by one. Together with induction over the natural number the above rule allows to express the required reachable properties.

For practical use, one defines convenience rules derived from the axiom insertion rule. These allow to treat often occurring special cases in an efficient way. Some examples for this kind of rules are listed below:

KeY

```
reachableDefinitionBase {
  \find(reach[...](t1, t2, 0))
  \replacewith(t1 = t2)
};

reachableDefinitionFalse {
  \assumes (n < 0 ==>)
  \find(reach[...](t1, t2, n)) \sameUpdateLevel
  \replacewith(false)
};

reachableDefinitionTransitive {
  \assumes (reach[...](t1, t2, n) ==>)
  \find(reach[...](t2, t3, m) ==>)
  \replacewith(reach[...](t1, t3, n+m) ==>)
};
```

KeY

3.3 Structural Specification of Graph Structures

The defined reachable predicate(s) are useful to express properties of general graph structures. In this section the specification of common linked structures using reachability is treated.

Some inconveniences that arise in the practise are due to the general nature of the predicate. Therefore further specification predicates are introduced and defined in terms of the reachable predicate. These predicates add an abstraction layer that simplifies the specification of (special) linked structures. In addition, it becomes possible to prove auxiliary lemmas for these predicates and to use them as taclets in proofs.

3.3.1 General Specification Predicates

In this section some general schemes for predicates are introduced. They are intended to be instantiated and reused for specification predicates dedicated to concrete data structures.

Definition 33 (Structure Constructors)**.** Given a linked data structure LS. The attributes or arrays of type LS building up the structure are called *structure constructors* of LS.

Example 9. Typical examples of structure constructors are:

- the attribute `next` for a typical single linked list as shown in Fig. 3.5.
- the attributes `left` and `right` for a binary tree like in Fig. 3.6.

Note 15. The definition restricts the considered structures LS nearly almost to inductively defined data structures, where a substructure is supposed to be itself an element of LS. The impact of the restriction for practical use is limited as many linked data structures are of this kind (maybe encapsulated by a wrapper class). The specification framework is flexible enough to treat implementations using a hierarchy of classes to model the graph structure's nodes as long as the hierarchy has one unique super class declaring the structure's constructors.

Characterisation of a Linked Data Structure In order to characterise a linked data structure LS one has to assign location descriptors[2], fix the concrete instance of the reachable predicate to be used, i.e., to concretise the *end* predicate and last but not least to define the characteristic predicate $\texttt{is}LS_{LS}(LS)$, which holds iff. the object given as argument represents an element of the structure.

Example 10. Common location descriptors for linked data structures:

Single Linked List. Assuming the structure constructor is an attribute called `next` the characteristic location descriptor is:

$$\texttt{\textbackslash for List sl; sl.next@(List)}$$

Binary Tree. Let the binary tree structure be constructed by attributes `left` and `right` referring to the corresponding children, then

$$\begin{array}{l}\texttt{\textbackslash for BinTree tr; tr.left@(BinTree);}\\ \quad\texttt{\textbackslash for BinTree tr; tr.right@(BinTree);}\end{array}$$

is the characteristic location descriptor.

Arbitrary Branching Tree or Graph. The following location descriptor

$$\texttt{\textbackslash for AB ab; int idx; ab.children@(AB)[idx]}$$

is the characteristic descriptor for arbitrary branching trees or graphs, whose children are kept in an array.

[2] in the following the linked data structure LS and its assigned location descriptors are identified, i.e., LS is used to denote both

For sake of readability the following abbreviations are used for the reachable symbol (and similar for other symbols of this kind)

- $\texttt{reach}_{LS}(start, end, step)$ for $\texttt{reach}[LS](start, end, step)$
- $\texttt{reach}_{LS}(start, end)$ for \exists *int step*; $\texttt{reach}_{LS}(start, end, step)$

For the definition of the characteristic predicate it has been proven helpful to build up on the following concepts:

Cycles $\texttt{onCycle}_{LS}(x) :\Leftrightarrow$ \exists *int* n; $(n > 0$ & $\texttt{reach}_{LS}(x, x, n))$

Finite Data Structure expressing that only a finite number of different elements can be reached from a certain point can be specified as follows:

$$\begin{array}{l}\texttt{finite}_{LS}(x) :\Leftrightarrow \\ \quad \text{\exists } LS[]\ elements;\ \text{\exists } int\ max;\ \text{\forall } LS\ z; \\ \quad\quad (\text{\exists } int\ d;\ \texttt{reach}_{LS}(x, z, d) \text{ -> } \\ \quad\quad\quad \text{\exists } int\ pos;\ (pos <= max\ \&\ elements[pos] \doteq z))\end{array}$$

In case the finiteness property is needed for an acyclic data structure there is a more convenient formalisation, which implies also the absence of cycles:

$$\begin{array}{l}\texttt{finite}_{LS}(x) :\Leftrightarrow \text{\exists } int\ max;\ \text{\forall } LS\ z; \\ \quad (\text{\exists } int\ d;\ \texttt{reach}_{LS}(x, z, d) \text{ -> } (d <= max))\end{array}$$

Unique Path In order to express the property that there is a unique path between two instances of a linked structure, one defines first the following helper predicate:

$$\begin{array}{l}\texttt{uniquePathAux}_{LS}(x, y, m) :\Leftrightarrow \\ \quad m >= 0\ \&\ \texttt{reach}_{LS}(x, y, m)\ \& \\ \quad (\bigwedge_{l_i \in LS}((\texttt{reach}_{LS}(x.l_i, y, m-1) \text{ -> } \\ \quad \texttt{uniquePathAux}_{LS}(x.l_i, y, m-1))\ \&\ \bigwedge_{j\ !\doteq i} !\texttt{reach}_{LS}(x.l_j, y, m-1)))\end{array}$$

On top of this predicate the intended predicate can be defined:

$$\begin{array}{l}\texttt{uniquePath}_{LS}(x, y) :\Leftrightarrow \text{\exists } int\ m;\ (\texttt{uniquePathAux}_{LS}(x, y, m)\ \& \\ \quad (\text{\forall } int\ m';\ (\texttt{uniquePathAux}_{LS}(x, y, m') \text{ -> } m \doteq m')))\end{array}$$

In addition there are:

$\texttt{in}LS_{LS}(n, s)$ is a binary predicate used to query, if the given structure n occurs as substructure of structure s. It is defined as

$$\texttt{in}LS_{LS}(n, s) :\Leftrightarrow \texttt{is}LS_{LS}(s)\ \&\ \texttt{is}LS_{LS}(n)\ \&\ \texttt{reach}_{LS}(s, n)$$

$\texttt{separate}_{LS}(s_1, s_2)$ denotes that both given structures are separate, this means do not share any node. Usually this is too strict as one sometimes want that certain elements may be shared (e.g. the empty list or null) etc. Therefore the given template parametrises $\texttt{separate}_{LS}$ with a predicate (a syntax notion already introduced in [Rot06]) that allows to specify an exceptional set of nodes that are allowed to be shared:

$$\texttt{separate}_{LS}^{\phi_{shared}(x)}(s_1, s_2) :\Leftrightarrow$$
$$\texttt{\textbackslash forall } LS\ x;\ ((\texttt{inLS}_{LS}(x, s_1)\ \&\ \texttt{inLS}_{LS}(x, s_2)) \texttt{ -> } \phi_{shared}(x))$$

In case of $\phi_{shared}(x) := false$ any sharing is prohibited, whereas for $phi_{shared}(x) := x \doteq \text{null}$ sharing of null is allowed.

Some typical properties derived from the above predicates are

Substructure Invariance. A direct consequence of the definition is:

$$\texttt{inLS}_{LS}(s, t) \texttt{ -> } \texttt{isLS}_{LS}(s)\ \&\ \texttt{isLS}_{LS}(t)$$

The property is not surprising at all, but having rules exploiting this fact makes proving easier.

Update Simplification. Knowledge about separate parts of linked structures allows to provide and apply update simplification rules to eliminate preceding updates that do not affect a particular (sub-)structure. These rules follow the principle idea:

$$\{\mathcal{U}\}\ \texttt{isLS}_{LS}(s) \Leftrightarrow \texttt{\textbackslash forall } LS\ x;\ l_i@pre(x) \doteq x.l_i \texttt{ ->}$$
$$\{\mathcal{U} \,||\, \texttt{\textbackslash for } x;\ \texttt{\textbackslash if(!}\,\texttt{inLS}_{LS}(x, s)\texttt{)};\ x.l_i := l_i@pre(x)\}\ \texttt{inLS}_{LS}(s)$$
resp.
$$\texttt{\textbackslash forall } LS\ x;\ \texttt{\textbackslash forall } int\ j;\ l_i@pre(x) \doteq x.l_i[j] \texttt{ ->}$$
$$\{\mathcal{U} \,||\, \texttt{\textbackslash for } x;\ \texttt{\textbackslash if(!}\,\texttt{inLS}_{LS}(x, s)\texttt{)};\ x.l_i[j] := l_i@pre(x, j)\}\ \texttt{inLS}_{LS}(s)$$

where

- $l_i \in LS$ occurs as part of $\texttt{\textbackslash for}\, LS\ x; \texttt{\textbackslash if}(\phi_i(x)); x.l_i$ resp. $\texttt{\textbackslash for}\, LS\ x;\ int\ j;\ \texttt{\textbackslash if}(\phi_i(x, j));\ x.l_i[j]$
- $l_i@pre$ is a new (not yet used) rigid function of the required arity

These rules allow further simplifications of specification predicate preceding updates. They make use of the last one wins semantic of updates where $\{x.a := c \,||\, x.a := x.a\}\ \psi$ can be simplified to $\{x.a := x.a\}\ \psi$ and finally to ψ.

3.3.2 Linked Lists

This section demonstrates how to use the reachable predicate and the construction schemes given in the previous section in order to provide a set of predicates and rules for the specification and verification of structural properties of a typical single linked list implementation (see Fig. 3.5).

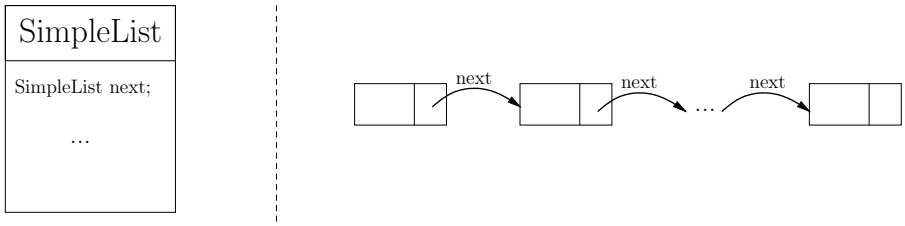


Figure 3.5: Single Linked List

Note 16. The specification predicates explained in the subsequent rely only on the constructors of the structure in this case the attribute function `next`. As long as the examined structure is solely build up using this constructor it is allowed to have e.g. subclasses of the `SimpleList` to mark special list elements. In particular it is also possible that the single linked list structure is embedded in a more complex structure.

The reachable predicate for single linked lists. The canonical location descriptor for a single linked list is

$$\backslash\text{for } List\ l;\ l.next@(List)$$

here:

$$SLL := \backslash\texttt{for SimpleList n;n.next@(SimpleList)}$$

In a first step, the end marker of the list structure needs to be defined. There are two commonly used designs:

- a node denotes the end of a list, if and only if the `next` attribute refers to `null`. In this case the end marker can be defined as:

$$end(x) :\Leftrightarrow x \doteq \texttt{null}$$

- there is a special instance used as terminator, e.g. `NIL` often accessible as final static field of the corresponding class. Thus:

$$end(x) :\Leftrightarrow x \doteq \texttt{SimpleList.NIL}$$

Therefore the reachable predicate of this structure is:
$\texttt{reach}_{SLL}(start, end, n) :\Leftrightarrow$
$(start \doteq end\ \&\ n \doteq 0) \mid (n > 0\ \ \&\ \ !end(start)\ \ \&$
$\backslash\text{exists } T\ listEl;\ (\texttt{reach}_{SLL}(listEl, end, n-1)\ \&\ start.next \doteq listEl))$

Note 17. Please note that the existential quantifier in the definition above has only one possible solution. Therefore one can eliminate the quantifier and use the simplified formula. In fact, that is done in practice. The simplified rule can be proven sound wrt. to the original axiomatisation within the KeY system.

The $\texttt{isSLL}_{SLL}$ predicate for a single linked coincides nearly completely with its reachable definition and can be defined as:

$\texttt{isSLL}_{SLL}(start) :\Leftrightarrow$
\exists SLL end; \exists int n; ($\texttt{reach}_{SLL}(start, end, n)$ &
$end \doteq SimpleList.NIL$)

On top of the above reachable predicate, it is convenient to define that the above structure implements a finite and acyclic list. The instantiated specification property for an acyclic list is

$\texttt{acyclic}_{SLL}(x) :\Leftrightarrow \texttt{finite}_{SLL}(x) :\Leftrightarrow$
\exists int max; \forall $SimpleList$ z;
(\exists int d; $\texttt{reach}_{SLL}(x, z, d)$ -> (d <= max))

The following short example shall serve as a motivation why the definition of these specification properties are not only convenient for specification purpose, but also allow while proving: An often experienced situation, when verifying programs iterating through linked data structures is:

```
reach[..](s.next,s,m), reach[..](s,s.next,n), ... ==> ...
```

Exploiting that reachability is a transitive relation one derives directly that

```
reach[..](s,s,m+n) ... ==> ...
```

must hold. If the considered structure is now known to be acyclic, one can use the following derivable rule

KeY

```
noCycles {
  \assumes(acyclic[..](s) ==>)
  \find(reach(s,s,m) ==>)
  \add(m=0 ==>)
}
```

KeY

which–when applied–leads to considerable simplifications. In many case the considered proof branch can be closed within a few steps.

3.3.3 Tree Structures

This section demonstrates how to compose a set of predicates and functions capturing structural properties of binary (similar finite arbitrary branching) trees.

The structural specification predicates listed below require a binary tree data structure similar to the one shown in Fig. 3.6. Similar means that the tree structure must be constructed solely by two distinguishable attributes (here:`left` and `right`). One may notice that this allows also tree implementations, which extend class `TreeNode` in order to have distinct node types for special purposes, e.g. leaves, red-black nodes, etc.

The characteristic location descriptor for binary tree implementations is

$Tree :=$ \for Tree n;n.left; \for Tree n;n.right;

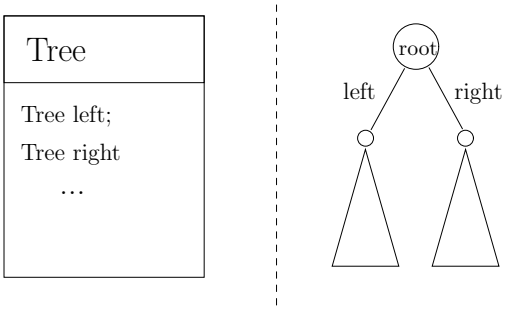


Figure 3.6: Binary tree implementation template

For the definition of the corresponding reachable predicate, one has to specify the end marker for paths, e.g.

$$\texttt{endMarker}_{Tree}(l) := l.\texttt{left} \doteq \text{null} \;\&\; l.\texttt{right} \doteq \text{null}$$

Usually the definition of the end marker coincides with the standard definition of leaves $\texttt{isLeaf}_{Tree}$. The next step is to define the predicate characterising, which elements are part of the structure:

$$\texttt{inTree}_{Tree} : \texttt{Tree} \times \texttt{Tree}$$
$$\texttt{inTree}_{Tree}(node, root) :\Leftrightarrow \backslash\text{exists } int\ d;\ \texttt{reach}_{Tree}(root, node, d)$$

A tree is defined as an acyclic graph with a unique path between the root and any node of the tree. The characterising predicate for a binary tree is then defined as

$$\begin{aligned}
&\texttt{isTree}_{Tree} : \texttt{Tree}\\
&\texttt{isTree}_{Tree}(t) :\Leftrightarrow\\
&\quad \backslash\text{forall Tree } n;\ (\texttt{inTree}_{Tree}(n,t) \texttt{->} (!\,\texttt{onCycle}_{Tree}(n) \;\&\\
&\qquad \backslash\text{forall Tree } m;\ \texttt{uniquePath}_{Tree}(n,m) \;\&\; \texttt{finite}_{Tree}(t)))
\end{aligned}$$

where the used auxiliary predicates are directly derived from the templates given in Sect. 3.3.1, i.e.,

- $\texttt{onCycle}_{Tree}(x) :\Leftrightarrow$
  $\backslash\text{exists } int\ dist;\ (dist > 0 \;\&\; \texttt{reach}_{Tree}(x, x, dist)$
- the $\texttt{uniquePath}_{Tree}$ predicate is defined as declared above. The necessary helper predicate is instantiated as follows:

$$\begin{aligned}
&\texttt{uniquePathAux}_{Tree}(x,y,m) :\Leftrightarrow\\
&\quad m >= 0 \;\&\; \texttt{reach}_{Tree}(x,y,m) \;\&\\
&\quad ((\texttt{reach}_{Tree}(x.left, y, m-1) \texttt{->}\\
&\qquad (\texttt{uniquePathAux}_{Tree}(x.left, y, m-1)\\
&\qquad\quad \& \,!\,\texttt{reach}_{Tree}(x.right, y, m-1))) \;\&\\
&\quad (\texttt{reach}_{Tree}(x.right, y, m-1) \texttt{->}\\
&\qquad (\texttt{uniquePathAux}_{Tree}(x.right, y, m-1)\\
&\qquad\quad \& \,!\,\texttt{reach}_{Tree}(x.left, y, m-1))))
\end{aligned}$$

- $\texttt{finite}_{Tree}(t) :\Leftrightarrow$
  \exists *int* *maxDist*; \forall `Tree` n;
  \forall *int* *dist*; ($\texttt{reach}_{Tree}(t, n, dist)$ -> $dist <= maxDist$)

Note 18. The finite predicate in the version used above implies already that the structure must be acyclic. Therefore the acyclic test can be skipped.

3.4 Summary

On the specification level recursive predicates provide a convenient way to specify linked data structures. It has been shown that using classical non-rigid predicates for this purpose has serious drawbacks concerning the achievable degree of automation.

Enabling the use of recursive non-rigid predicates while achieving a high degree of automation is the main contribution of this section. Therefore a variant of non-rigid symbols, so called *location-dependent* symbols, have been introduced. Location-dependent symbols come with a set of location descriptors attached describing all locations on which such a symbol may possibly depend. In other words, a location-dependent symbol has to be interpreted same in any two states which coincide on the value of the described locations.

It has been demonstrated how this kind of information leads to shorter proofs and allows to achieve a significant increase in automation. A proof obligation formula has been introduced for this kind of symbols which allows to ensure that their definition obeys the explicit dependency restrictions. The presented solution enables a set of powerful simplification rules for a whole class of symbols. In contrast to more specialised solutions for which auxiliary rules/lemmas needs to be proven separately for each new introduced predicate in order to achieve the same results.

Further, application areas in specification of linked data structures as well as their usage for modelling queries in the logic have been explored.

4 Recursive Methods Treatment

4.1 Motivation

The following sections focus on correctness proofs for recursively defined methods. In order to clarify the notion of a recursive method, it proves useful to introduce the notion of an execution trace:

Definition 34 (Symbolic Execution Trace)**.** A *symbolic execution trace* $Tr_S(st)$ is a not necessary finite list $(st_0(= st), \ldots, st_i, \ldots)$ of all statements to be executed when starting the symbolic execution of statement st in state S.

Note 19. As dealing with a deterministic language and starting in a defined state S it is always possible to represent the symbolic execution trace as a list in execution order. In order to capture all traces when starting in an unknown or only partial known state a tree (or more general) data structure has to be taken. A trace corresponds then to one path in the structure.

Note 20. Intuitively a symbolic execution trace captures the stepwise decomposition of a statement (or program) into atomic programs.

Example 11. The statement `i=(j=3)+1` executed in an arbitrary state S leads to the following symbolic execution trace:

$$Tr_S(\mathtt{i} = (\mathtt{j} = 3) + 1;) := (\mathtt{i} = (\mathtt{j} = 3) + 1;, \mathtt{j} = 3;, \mathtt{i} = \mathtt{j} + 1;)$$

It is now possible to define precisely the meaning of the term *recursive method*:

Definition 35 (Recursive Method; Re-entry point)**.** Let the symbols `o` and $\mathtt{v_i}$ ($i \in \mathbb{N}$) denote program variables (or at least side effect free expressions) of a fitting type. A method `m` is called *recursive*, if there is an execution trace $Tr_S(\mathtt{o.m}(\mathtt{v_1}, \ldots, \mathtt{v_n});)$ of `m` that contains another invocation of method `m` at a position $i > 0$. Position i is called a *re-entry point* of method `m`.

4.1.1 Current Problems and Challenges

This section explains the problems and limitations of the current KeY system when a recursive method (or re-entry) occurs along a simple example. The following recursive implementation of a simple method computes the factorial of a given number:

```
public class MathLib {

   public static int fac(int n) {
```

```
        if (n==0 || n == 1) {
            return 1;
        } else {
            return n*fac(n-1);
        }
    }
}
```

The factorial is not expressible in terms of a closed arithmetical expression. Instead a proper specification may define a recursively defined (logic) function fac according to the common mathematical definition:

KeY

```
factorialDefinition {
  \find (fac(x))
  \replacwith(\if ( x > 1 ) \then (x*fac(x-1))
       \else ( \if (x>=0 & x<=1)
                \then (1)
                \else fac(x) ) )
};
```

KeY

In order to prove the library function `fac` to be equivalent to the function fac for non-negative arguments. In a first approach we try to prove

KeY

```
   \forall int x; (x>=0 -> {i:=x}
           \<{ result = MathLib.fac(i); }\> result  = fac(x))
```

KeY

using the standard integer induction rule. The induction hypothesis can be chosen straight forward. The base and use case can be closed easily, but the step goal cannot be closed although valid. The remaining open goal looks like:

KeY

```
 nv_1 >= 1,  x_1 >= 0,
 {n := nv_1} \<{ { j = MathLib.fac(n)@MathLib; }
                   result = j; }\> fac(nv_1) = result
==>
 {j_2:=1 + nv_1 || n_1:=nv_1}
   \<{ {method-frame(result->j, source=MathLib)
       : { { j_4 = MathLib.fac(n_1)@MathLib; }
           j_3 = j_4;
           j_1 = j_2*j_3;
           return j_1;
         }
       }
       result=j;
     }\> fac(1 + nv_1) = result
```

KeY

Although the result of the method invocation is in some sense given by the third formula in the antecedent, the proof stops at this point. The reason is the induction hypothesis is not general enough. The root of the problem is that it is not possible to quantify about the method stack in the used logic.

Instead of introducing some instance of a higher order construct to quantify/generalise about the method stack (and therewith about programs), the rule presented in the next section uses taclets to simulate this kind of quantification. The taclet language allows to generalise in the above sense by using the schematic program context $..(\cdot)...$. The schematic program context allows to match the inactive program prefix $(..)$, the active statement $((\cdot))$ and the rest of the program $(...)$. In fact it represents a possibility to quantify about all programs that contain the specified statement $(\cdot)$.

There are (at least) two alternatives to present the taken approach. The first alternative introduces a new taclet (rule) that allows to prove the validity of a property ϕ in presence of a recursive method. This means, in the case the a proof of ϕ has to consider a recursive implemented method.

The second alternative describes how to generate a set of proof obligations that allow to prove the correctness of a recursive method with respect to a given contract. Afterwards, the proven contract can be used when proving the property ϕ.

The second alternative allows a more concise explanation of the principal idea as it has to cope with less technical problems. Therefore this alternative is chosen to present the approach. The first alternative will be discussed in a later section.

4.2 Recursive Method Treatment

4.2.1 Using Proof Obligations

Two(+1) proof obligation are required to prove the correctness of a recursive method with respect to a given contract $C := (pre, post, mod)$, instead of one for non-recursive methods.

The principle idea is to combine induction and the method contract rule. The first proof obligation to be generated represents the base case of the induction. In this context, the base case contains usually exact those cases for which the recursion can be eliminated by a (small) finite number of method body expansions. A non-negative distance function $dist$ is used to characterise exactly those states that belong to the base case. That are all states in which the distance function evaluates to 0.

The second proof obligation includes a restricted variant of the contract C in its rules collection. Restricted means that the precondition is stronger in the sense that it requires the distance function to have a value lesser than an arbitrary but fixed value.

In the third (+1) proof obligation, it has to be shown that the formula $dist > 0$ holds in any state that satisfies the method's precondition pre.

It is necessary to loose some more words about the provable method contracts C. The contract to be provable following this approach has to be in general

more detailed than one would expect or usually specify in JML or OCL. The reason for this is that during the execution of the recursive method one may observe intermediate states at the re-entry points which could not be observed by an outside caller. This means that at invocation time one may not necessary assume that all invariants are valid and will have thus to encode the required parts of the invariant in the precondition. Taking special care of invariants is necessary, as we use in KeY an observable state semantics for invariants, in contrast to JML's visible state semantics. In addition, it may be necessary to relax the precondition itself for certain re-entry points.

Furthermore the modifies set has to include more locations as a temporarily changed location may be restored only some time after returning from the re-entry point and thus changed field values may be observable. It should be noted that the locations contained in the modifies set can be further restricted to static or instance fields and need not to take local fields into account.

If the method contract becomes too complex for practical purposes, it can be used afterwards to prove a stronger contract C_{pub} better suited for general (public) use.

Some preliminary definitions are still required, mainly the notion of a *recursive method contract*:

Definition 36 (Recursive Method Contract; Distance Function)**.** A *recursive method contract* $RecCtrt := (pre, post, mod, dist)$ extends a normal method contract by a distance function $dist$, which is specified as an integer typed term that has to evaluate to a non-negative value in any state satisfying the precondition pre.

In the next paragraphs the following abbreviations will be used:

- The symbols $\overrightarrow{args} := (arg_1, \ldots, arg_n), \overrightarrow{argslv} := (arglv_1, \ldots, arglv_n)$ represent sequences of program variables (respective logical variables) of equal size which have pairwise the same type, i.e., $sort(args_i) = sort(argslv_i)$. This notation is typically used to abbreviate quantification or updates, e.g.
  - $\forall \overrightarrow{argslv};$ is short for $\forall arglv_1 : T_1; \ldots; \forall arglv_n : T_n;$
  - $\overrightarrow{args} := \overrightarrow{argslv}$ is short for $arg_1 := arglv_1 \,||\, \ldots \,||\, arg_n := arglv_n$
- The expression $dist(\overrightarrow{args})$ is an integer typed term representing the distance functions. In order to make the dependence of $dist$ on the method's arguments more explicit they are mentioned explicitly. It does not mean that $dist$ must depend on all arguments or only on those arguments.
- The non-rigid predicate po_{mod} is an abbreviation for the proof obligation of the modifies set.
- The program variable $self$ is used to address the receiver of the method call. It can be used by the pre- and postcondition.

For sake of simplicity treatment of exception will be skipped. At a later point it will be explained how to incorporate abrupt termination of methods

by uncaught exceptions. The way exceptions are supported is equivalent to the approach taken for standard method contracts.

Base Case The proof obligation to be generated for the base case is

$$\Longrightarrow \forall o{:}T;\ \forall \overrightarrow{argslv};\ \{self := o \,||\, \overrightarrow{args} := \overrightarrow{argslv}\}$$
$$(o \doteq \text{null}\ \&\ pre\ \&\ \ dist(\overrightarrow{args}) \doteq\ 0 \texttt{ -> } (\langle self.m(\overrightarrow{args})@T;\rangle(post)))$$

Most parts of the formula are identical to the standard proof obligation for method contracts. As usual the adherence to a given contract is shown for a fixed implementation of method `m`, namely the implementation found in type T. Thus the concrete method body statement of this implementation is used in the modality.

In order to prove the validity of the formula one has to show that for any non-null object o on which the method is invoked and any parameter assignment $\overrightarrow{args} := \overrightarrow{argslv}$ the method satisfies the contract's postcondition after its execution (and that the method terminates). But only if the precondition pre holds and the distance function evaluates to 0. The latter point is where the generated proof obligation deviates from the contract rule for non-recursive methods.

Example 12. The proof obligation for the factorial example is:

KeY

```
\forall int inlv;{in:=inlv}(in >= 0 & in = 0 ->
   \<{ result=MathLib.fac(in)@MathLib; }\>(result = fac(in)))
```

KeY

which can be closed in a couple of steps and one interaction namely the application of the taclet factorialDefinition. The quantification about all method call receivers is missing, because the method is declared static.

Step Case The formula of step case proof obligation differs only slightly from the one of the base case:

$$\Longrightarrow_{\cup mc_{restricted}} \forall o{:}T;\ \forall \overrightarrow{argslv};\ \{self := o \,||\, \overrightarrow{args} := \overrightarrow{argslv}\}$$
$$(o \doteq \text{null}\ \&\ pre\ \&\ \ depth >=\ 0\ \&\ dist(\overrightarrow{args}) \doteq\ depth + 1 \texttt{ -> }$$
$$(\langle self.m(\overrightarrow{args})@T;\rangle(post)))$$

where $depth$ is a newly introduced rigid integer typed constant.

The only visible difference is that the distance function is assumed to evaluate to value $depth+1$, where $depth >= 0$. The integer induction rule would provide a formula in the premise stating that the correctness of the method contract has already been proven for all cases where $0 <= dist(\overrightarrow{args}) <= depth$. As already explained, that is not expressible on the syntactical level of our logic.

Instead the proof obligation for the step case contains an additional rule $mc_{restricted}$[1], namely the application rule for the method contract to be proven,

[1] in fact there are two rules, one for the antecedent and one for the succedent

but with one important derivation: In order to apply the method contract, it has not only to be shown that the precondition is satisfied, but also that the distance function (or term) evaluates to a non-negative value lesser or equal than *depth*.

Example 13. The proof obligation for the factorial example is:

KeY

```
\forall int inlv;{in:=inlv}((in>=0 & depth>=0 & in=depth+1) ->
         \<{ result=MathLib.fac(in); }\>result = fac(inlv))
}
```

KeY

The restricted version of the method contract can be written in pseudo taclet code as

KeY

```
methodContractRestricted {
 \find (==>
    \<{.. #result = MathLib.fac(#v)@MathLib; ...}\>(phi))
 "Precondition␣and␣Restriction":
    \add ( ==>  #v>=0 & depth >= 0 & #v <= depth  ) ;
 "PostCondition" :
    \replacewith (==> #result=fac(#v) -> \<{..  ...}\>(phi))
};
```

KeY

The proof has been performed in the KeY system. The closed proof tree consisted of 231 nodes on three branches. Four interactive steps have been required (1×methodBodyExpand, 1× methodContractRestricted, 2×factorialDefinition).

Well-Defined Distance Function After proving the last two proof obligation, one has shown that the method satisfies its contract, but only when the distance function evaluates to a non-negative value. Either we can add this requirement to the precondition and show that–at any time–when the method contract is used or one can try to get rid of the distance function. The last proof obligation serves the latter approach, but it may (in rare cases) make the formulation of the precondition or distance function more complicated.

The third proof obligation

$$\Longrightarrow \forall o{:}T;\ \forall \overrightarrow{argslv};\ \{self := o \,||\, \overrightarrow{args} := \overrightarrow{argslv}\}\ ((o \doteq \text{null}\ \&\ pre) \rightarrow depth >= 0)$$

is usually the easiest to prove. It states that in any state satisfying the precondition, the distance function term must evaluate to non-negative value.

Example 14. For the running example, we get

KeY

```
\forall int inlv;{in:=inlv}(in>=0 -> in>=0)
```

KeY

which can be nearly immediately proven.

Correctness of the Modifies Set It remains to show the correctness of the modifies set. The procedure is analogously to the one above, i.e., the modifies set is first shown to be correct for all states where *dist* evaluates to 0.

The step case assumes again that the modifies set is correct for all method invocations in a state satisfying the precondition and where the distance function evaluates to an arbitrary, but fixed non-negative value. The restricted version of the method contract is added again to the rule base.

The well-definedness case remains unchanged.

The proof obligation to be generated is the one shortly described in Sect. 2.2.6 and the first time introduced in [Rot06]. Besides adding the distance function requirements to the different proof obligations nothing has to be changed.

Example 15. The factorial computation has an empty modifies set, this means in order to fulfil the contract it must not change the value of any instance or static field. The proof obligation for the step case is:

KeY

```
\forall int inlv;{in:=inlv}((in>=0 & depth>=0 & in=depth+1) ->
   ((\<{ result=MathLib.fac(in); }\>
        \<{ MathLib.anon()@MathLib; }\>true) <->
             \<{ MathLib.anon()@MathLib; }\>true))
```

KeY

In addition with the restricted method contract rule (for the antecedent and succedent). The closed proof has 305 nodes, 6 branches and required four interactive steps (2×methodBodyExpand, 2×methodContractRestricted)

4.2.2 Example: List Reversal

In order to demonstrate how to verify a recursive method contract, we reuse the `SimpleList` implementation of a single list once again. The task is to prove that the recursive instance method contract of the method `reverse()` is correct. The method implementation

```
public void reverse() {
   if (next != null) {
     next.reverse();
     next.next=this;
     next = null;
   }
}
```

is obviously recursive. It descends until the end of the list has been reached. Up to this point the list structure has not been altered yet. Then the method returns step-by-step and after each ascending step, it reverses the `next` reference of its next element. For example, assume the following sequent to represent the list $(n_0, \ldots, n_{sz})$, when the algorithm has reached node n_{sz}, it starts returning.

After taking one step upwards, it is back at node n_{sz-1} and the list still unchanged. Now it redirects the `next` reference of node n_{sz} to the current node n_{sz-1} and so further on until n_0 is reached.

The property to be proven is that when executing `self.reverse()` the method terminates normally and afterwards

- the `next` reference of the first element `self` is **`null`**.
- for any element e other than `self`, the `next` reference has been reversed to its predecessor, i.e., the equation e`.next.next@pre`$\doteq e$ is satisfied (the unary function `next@pre` captures the value of `next` in the pre-state for any list).

The specification of the method contract uses the linked data structure predicates as introduced in Sect. 3.3. The contract to be verified can then simply be written as

```
let next@pre(list:SimpleList) = list.next in
let elementInList:SimpleList such that \exists n;
       reach[\for (SimpleList l)
                 l.next@(SimpleList)](self,x,n) in

pre:
  self != null &
  \forall SimpleList list;
      isList[\for (SimpleList l) l.next@(SimpleList)](list)

post:
   \if (elementInList.next = null)
   \then (elementInList = self)
   \else (nextAtPre(elementInList.next) = elementInList)

distance:
   (\ifEx (int len)(len>0 & reach[\for (SimpleList l)
         l.next@(SimpleList)](self, null, len)) \then (len-1)
                                                \else (-1))
```

The let-definitions as defined in Sect. 2.2.6 are evaluated in the pre-state. The declared function symbols like `next@pre` have to be created newly for each application. The declared logic variable `x` to represent an arbitrary element of the list, will be bound and is therewith always locally declared (i.e., newly introduced).

In the above contract the first let definition introduces the new rigid function symbol `next@pre` used to capture the pre-state value of function `next`. The second let definition fixes an arbitrary element of the list starting at `self`.

The distance term is defined using the \ifEx x T; operator. This operator binds a logical variable `x` in its guard and its \then branch. The term

evaluates to the value of its \then branch, if there exists a smallest element that satisfies the guard formula[2] and otherwise the value of the term is equal to the value of its \else branch.

For a list of length *size* the distance term evaluates to $size - 1$. Note, that lists of type `SimpleList` are **`null`** or have at least one element.

In the next paragraphs, the correctness proof of this method contract is shortly sketched and a short statistic is given. The number of interactive steps is higher than necessary, i.e., the linked structure predicates are not treated automatically at all. Further, finding the shortest proof has not been one goal, i.e., in particular there are unnecessary interactive steps, which have been just applied for convenience reasons during the proof, e.g. to get the sequent more readable.

Base Case: Lists with one element. For the base case we have to prove that the method satisfies the contract for all lists with exact one element. The generated[3] proof obligation is:

KeY

```
\forall SimpleList selfLV;{self:=selfLV}(
  \forall SimpleList list;(nextAtPre(list) = list.next) ->
    \forall SimpleList elInList;(\exists int idx;
      (elInList != null & idx >= 0 &
         reach[\for (SimpleList l)
                 l.next@(SimpleList)](self,elInList,idx)) ->
// precondition
 ( ( self != null &
     \forall SimpleList list;
       isList[\for (SimpleList l) l.next@(SimpleList)](list) &
// distance restriction
      (\ifEx (int len)(len>0 & reach[\for (SimpleList l)
                          l.next@(SimpleList)](self, null, len))
       \then (len-1) \else (-1)) = 0
    ) ->
    \<{ self.reverse()@SimpleList; }\>
// postcondition
       (\if (elInList = self)
        \then (elInList.next = null)
        \else (nextAtPre(elInList.next) = elInList))
  )))
```

KeY

[2] In fact the logic used in KeY defines a well-order on the universe. Consequently, if a smallest element e exists that satisfies a given formula $\phi(e)$, there exists always a smallest element m such that $\phi(m)$.

[3] At the moment of writing the generation is done manually. The generation will become automatically after the new contract framework for methods has been introduced.

The first four lines are the translation of the contracts `let` constructs. Followed as marked by the translation of the precondition. The precondition has been strengthened and requires that the value of distance term is equal to zero. Finally, the translated postcondition to be shown satisfied is placed directly after the modality that contains the method invocation.

The base case branch as well as all the other branches have been proven in the KeY system. The closed proof for the base case branch consists of 135 nodes on four branches, twelve interactive steps have been applied. The proof is very simple, the greatest part of the work had been investigated to derive from the distance function that `self.next` is **`null`** and thus no recursive call occurs.

Step Case: Lists with more than one element. As expected the step case is the interesting one. The generated proof obligation assumes that the contract has been already proven for lists with less than `depth` elements. For those lists the restricted method contract(see Fig. 4.1) is already available.

The proof obligation itself is equal to the base case proof obligation except for the distance term, which covers this time all lists for which the distance term evaluates to `depth+1` for some arbitrary value `depth` with `depth>=0`.

The proof fixes an arbitrary list node `elInList0` of the list starting at element `self`. Then one expands the method `reverse` exactly once.

When the re-entry point is reached, that is the recursive call of `reverse` on the successor node `self.next`, one applies the restricted method contract. At this point two cases need to be distinguished and treated differently. Depending on whether the element of interest `elInList0`

1. is the same as `self`: In order to close this goal, the only assertion of the method contract that can (and must) be used, is that the method terminates. Therefore we instantiate the let variable definition of the method contract with `self.next` and have later on only to show that the precondition and distance term restriction holds. The goal can be closed by symbolic execution of the remaining program after the recursive method call.

2. the `self.next`: This subgoal can be closed when instantiating the method contract let construct with `elInList0`. After showing that the precondition and distance restriction is satisfied once again, all that is needed to close the goal is asserted by the used method contract's postcondition.

The closed proof for the step case has 963 nodes on 24 branches and required 87 interactive steps.

4.3 Summary

This section introduced a rule that allows to prove that the implementation of a general recursive method satisfies a given specification in a dynamic logic with state updates and an implicit method call stack as used in JAVA CARD DL. The main idea is to combine the induction rule and contract application rule.

```
restrictedMethodContract {
 \schemaVar \variables SimpleList l, list1, list2;
 \schemaVar \variables int idx, len, n;
 \schemaVar \program Variable #o, #self;
 \schemaVar \program MethodName #mn;
 \schemaVar \formula phi;

 \find(==> \<{.. #o.#mn(); ...}\> phi)
 \varcond(\new(#self, \typeof(#o)),
          \notFreeIn(l, phi), \notFreeIn(idx, phi),
          \notFreeIn(list1, phi), \notFreeIn(list2, phi))
 \replacewith(==>
   {#self := #o} (
// let definitions
     \forall list1;(nextAtPre2(list1) = list1.next) ->
       (\exists l;((\exists idx; (l!=null & idx>=0 &
          reach[\for (listLV)listLV.next](#self, l, idx)))
       &
// precondition
        (
          #self != null
        &
          \forall list2; isList[\for (listLV) listLV.next](list2)
        &
// distance restriction
         depth >= 0
        &
         (\ifEx (len)(len>0 &
            reach[\for (listLV)listLV.next](#self, null, len))
          \then (len-1) \else (-1)) = depth
        )
// postcondition
       & {\for list2; \if (\exists idx; (idx>=0 &
            reach[\for (listLV)listLV.next](#self, list2, idx)))
              list2.next:=anon(list2)}
         (
           (\if (l = #self) \then (l.next = null)
                            \else (nextAtPre2(l.next)= l))
           -> \<{.. ...}\>(phi))))))
 \addprogvars(#self)
};
```

Figure 4.1: Restricted Method Contract for List Reversal

Allowing the restricted use of the contract to be proven in the step case paying attention not to introduce a circular dependency. The presented approach has been explained and evaluated using two examples: Fibonacci numbers and list reversal.

5 Specifying Linked Data Structures with Abstract Data Types

5.1 Abstract Data Types and Linked Data Structures

Abstract data types are a well-known and explored area in computer sciences. Of interest for this section are generated abstract data types as they enable reasoning by structural induction.

Generated abstract data types are axiomatised with help of a set of function symbols. A selected subset of them is marked as constructors. Their semantics is that all elements of the abstract datatype can be written as ground terms consisting only of constructors. In particular any element of the domain has a syntactical representation. The remaining functions can be divided into observers, which do not manipulate the structure, and operations mapping an ADT element to another one.

The semantics of the queries and operations is usually given in terms of a rewrite system and/or general axioms in a (fragment) sorted first order logic. Some characteristics of the algebraic data type are strongly tied to the allowed rewrite system resp. logic.

Abstract data types are a useful tool to model linked data structures. To some extend they allow to (re-)use methodologies invented for their analysis for verifying object-oriented programs.

5.1.1 Abstraction of Linked Data Structures

This section demonstrates how to abstract a linked data structure using abstract data types. The found abstraction can then be used instead of the linked data structure itself when verifying programs *using* the data structure. An elaborated example is given in Sect. 6 modelling the JAVA String data type in KeY.

An abstract data type is traditionally described using an algebraic specification. An algebraic specification defines a signature algebra consisting at least of

- a set of sorts (sort names) $\mathcal{T}_{ADT}$ with one distinguished $S \in \mathcal{T}_{ADT}$ often called the sort or type of interest.
- a set of functions F_{ADT}
- a set of axioms Ax_{ADT}

The intent of the axioms set is to give the functions a meaning. The axioms are usually given as (conditional) equations. The concrete used framework is

characteristic for the expressiveness of the specified algebra. It ranges from (universally closed) equations to conditional equations or a full first-order logic. While the first often allow a direct execution of the specification emulated by a rewrite system, this is in general not possible for the more powerful logic languages.

The semantics of an algebraic equation is then given as an interpretation assigning each sort a domain, the function symbols a mathematical function such that the axioms are valid. An interpretation of this kind is called *model of the algebra*.

The close relation ship to our specified logic is evident in particular as the logic itself can be seen as an algebra itself.

Definition 37 (Abstract Data Type)**.** A collection of interpretations of an algebraic specification is called *abstract data type* if and only if it is closed under isomorphism.

Many algebraic specification languages allow to specify additional constraints on the set of interpretations to be considered. Those constraints cannot be expressed as axioms in a first order language. The interesting constraint for this theses is called *generated*, others are for example freely generated, loose etc.

Definition 38 (Generated ADT)**.** An abstract data type is called *generated*, if there is a set $C \subset \mathcal{F}_{ADT}$ such that any element, belonging exactly to the domain of the sort of interest, can be represented by a ground term consisting only of functions and constants (nullary functions) in C.

Specifying an abstract data type in JAVA CARD DL and the KeY system is straight forward: one declares the sorts and (rigid) functions of the ADT and adds the axioms as rules or universally closed formulae to the sequent. The way how the axioms are formalised has some influence on the proving style and degree of automation. Adding them as formulae (or nearly equivalent as rules, which add these formulae to the antecedent of a sequent) requires in general the application of instantiation rules and result in a declarative but less automatic proving style. The taclet formalism allows in particular to draw benefit from algebraic specifications using (conditioned) equations. These can be easily translated into rewrite taclets with an optional assumes part containing the rewrite rule condition.

Example 16. Given axiom `ax` as conditioned equation

$$x = 0 \texttt{ -> } 5 * x \doteq 0$$

where x is an implicit universally bound variable. From this one can construct the taclet

```
ax {
 \assumes (==> x = 0)
 \find (5*x)
 \replacewith (0)
};
```

It is also possible to create a symmetric taclet where the find and replacewith parts are swapped. But then the character of a directed equation is lost and therewith an (easily) executable specification.

If a given abstract data type has to be generated, a structural induction rule is added as taclet to complete the axiomatisation. The structural induction rule for a data type T is a cut rule of the following shape:

$$\text{structInduction}\quad \frac{\text{Base Case}_{i=1}^{|C_0|} \qquad \text{Step Case}_{i=1}^{|C-C_0|} \qquad \text{Use Case}}{\Gamma \Longrightarrow \Delta}$$

- for all $c_i \in C_0$: Base Case$_i$:= $\Gamma \Longrightarrow Cl_\forall(\{\backslash\text{subst } T\ iv;\ c_i\}IH),\ \Delta$
- for all $c_i \in C - C_0$ let $\alpha_T(c_i)$ denote the number of arguments of c which are of type T. Then Step Case$_i$:=

$$\Gamma \Longrightarrow \begin{array}{l} \backslash\text{forall } T\ nv_1, \ldots, nv_{\alpha_T(c_i)}; \\ \quad ((\bigwedge_{i=1}^{\alpha_T(c)} Cl_\forall(\{\backslash\text{subst } T\ iv;\ nv_i\}IH)) \rightarrow \\ \qquad Cl_\forall(\{\backslash\text{subst } T\ iv;\ c_i(nv_1, \ldots, nv_n)\}IH)),\ \Delta \end{array}$$

- Use Case := $\backslash\text{forall } T\ nv;\ \{\backslash\text{subst } T\ iv;\ nv\}IH$

with

- IH as induction hypothesis and free induction variable iv of type T
- $C_0 := \{c \in C | \alpha(c) = T_1 \times \ldots \times T_n \rightarrow T, T_i \neq T \text{ f.a. } i \in \{1 \ldots n\}\}$
- $Cl_\forall(\phi)$: universal closure of ϕ. Necessary, as the constructors may include other types as arguments, e.g. the content stored in one list element.

In Fig. 5.1 an excerpt of an algebraic specification of a linked data structure is shown. Its embedding in JAVA CARD DL is presented in Fig. 5.2.

5.1.2 Connecting Abstract Data Structure and JAVA CARD DL

While the previous section summarised how to model a linked data structure, this section describes how to connect them to linked data structures modelled in JAVA.

Therefore the following requirements need to be fulfilled:

- a relation ρ between both formalisations of the linked data structures has to be defined. Intuitively the relation shall hold if the abstract data type is a valid abstraction of the concrete one.
- operations modifying the linked data structure must be mapped to operations on the abstract data type.

```
sort AList import int
generated

constructors nil:AList | cat(int, AList)

prepend::AList -> AList -> AList
prepend(first, nil) = first
prepend(first, cat(x, tail)) =
          cat(x,prepend(first, tail))

get::AList -> int -> int
get(cat(x, fstTail), n) =
    if (n = 0) then
      x
    else
      get(fstTail,n-1)
    fi
```

Figure 5.1: Algebraic Specification of list data type `AList` (`first, tail, fstTail` are variables of type `AList`, `n,x` are variables of type `int`)

In order to model the connection between an abstract data type and the concrete linked data structure, the latter one is equipped with a 'virtual' attribute. This virtual attribute is modelled as a location function and behaves like a normal attribute with two exceptions:

1. location functions cannot occur as part of a program and
2. they can be declared of a sort, which is not derived from a program type like reference or primitive types.

Definition 39 (Location Function)**.** A *location function* is a non-rigid function, which can occur as top level symbol on the left side of an update's assignment pair.

Note 21. The name *location function* has been chosen to emphasise that the function models a location and therefore can be evaluated as a semantic location as defined in Def. 10. Consequently, it is directly updateable via an update.

Attribute and arrays are special kinds of location functions as they are allowed to occur in and therewith to be changed by programs.

In principle one could allow any non-rigid function to be modified directly by an update and would thus not need to define a new syntactic category. The drawback would be that the use of non-rigid predicates as auxiliary functions becomes more difficult as one could not define them via simple rewrite rules. For example, let the unary non-rigid boolean function `nonNullF` be intuitively defined to evaluate to TRUE, if the argument's value is not equal to the value of the `null` literal. In this setting, the rewrite taclet nonNullConcrete

KeY

```
\sorts {AList;}

\functions {
  AList nil, cat(int, AList); // constructors
  AList prepend (AList, AList), get(AList, int);
}

\rules {
 prependEmptyList {
  \schemaVar \term AList first;
  \find(prepend(first, nil))
  \replacewith(first)
 };

 prependNonEmptyList {
  \schemaVar \term AList first, sndTail;
  \schemaVar \term int x;
  \find(prepend(first, cat(x, sndTail)))
  \replacewith(cat(x, prepend(first, sndTail)))
 };

 getElement {
  \schemaVar \term AList fstTail;
  \schemaVar \term int x, n;

  \assumes (n>=0 ==>) // alternative: \add( ==> n>=0)
  \find(get(cat(x, fstTail), n))
  \replacewith(\if (n = 0) \then (x) \else (get(fstTail, n-1)))
 };

 // providing structural induction implies generatedness
 structInductionAList {
   \schemaVar \formula b;
   \schemaVar \variables AList nv;
   \schemaVar \variables int x;

   \varcond(\notFreeIn(x,b))
   "Base Case": \add(==> {\subst nv; nil}(b));
   "Step Case": \add(==> \forall nv ; (b ->
                           \forall x;{\subst nv; cat(x, nv)}b));
   "Use Case" : \add(\forall nv;b ==>)
 };
}
```

KeY

Figure 5.2: Java Card DL Specification of `AList`

```
KeY
nonNullConcrete { \find(nonNull(null)) \replacewith(FALSE) }
KeY
```

would not be sound, which becomes obvious when we look at the following formula

```
KeY
{nonNullF(null):=TRUE} nonNullF(null) = FALSE
KeY
```

Dependant on the rule application order, it is possible to derive TRUE when the taclet nonNullConcrete is applied before the update simplification rules, otherwise FALSE can be derived.

The kind and use of these functions is similar to concepts found in other specification languages like JML model fields or let-definitions in OCL. On advantage of their use is the connection of two worlds - programs and logic. It embeds abstract data types as primitive types into JAVA.

As attributes are in principal unary location functions, the update simplification rule for the latter one can be easily derived by a slight generalisation of the rule for applying updates on attributes:

$$\{\overbrace{g(y_1,\ldots,y_n) := v}^{:=\mathcal{U}}\}\ f(x_1,\ldots,x_n) \rightsquigarrow \begin{cases} v, \text{if } g = f \text{ and } val(y_i) = val(x_i) \\ f(\{\mathcal{U}\}\ x_1,\ldots,\{\mathcal{U}\}\ x_n), \text{ otherwise} \end{cases}$$

Let L denote a concrete linked data structure and L_A its abstraction and let the location function $\texttt{abstraction} : L \to L_A$ be used as the corresponding virtual attribute.

The intention is to establish the following property. Given an object l representing the concrete data structure and its abstraction l_A

$$\rho(l, l_A) :\Leftrightarrow \texttt{abstraction}(l) \doteq l_A \tag{5.1}$$

The difficulty when trying to ensure property 5.1 is that the virtual attribute may be altered to any value using an update possibly invalidating 5.1. This implies that one cannot simply assume that the value of the location function $\texttt{abstraction}(l)$ satisfies the above equation 5.1, but that it must be stated explicitly when one intends to rely on the equation. Let *po* denote a proof obligation, which makes use of the abstraction. In order to be sound one has to assume that all considered states represent which relate the linked data structure L of interest to a valid abstraction:

$$\backslash\text{forall } L\ l;\ (\rho(l, \texttt{abstraction}(l))) \texttt{ -> } po$$

5.1.3 Applications

Specifying (collection) libraries. The advantage of relating an instance of a linked data structure to an instance of an abstract data type becomes immediately when specifying libraries like collection libraries. The operational semantics of the methods provided by the interfaces (and classes) of these libraries

can be specified easily if one can refer to abstract data types. Of particular advantage is that most algebraic data type specification are at least to some point executable in a functional style. This leads to easy readable specifications and provides sufficiently different specification and implementation languages.

An excerpt of a specification of a `List` interface is shown below. The location function is named `<abstraction>`. In order to emphasise its close relationship to an attribute the remaining part of the chapter makes use of the postfix notation, i.e., `o.<abstraction>` instead of `<abstraction>(o)`. The location function `<abstraction>` maps an instance of type `List` to an instance of the abstract data type `AList`. The excerpt used an pseudo-OCL syntax to provide a better readability than using the JAVA CARD DL syntax:

OCL

```
context List::get(int idx):Object
pre  : idx >= 0 and idx < self.size()
post : result = get(self.<abstraction>, idx)
modifies: {}

context List::size():int
pre  : true
post : result = size(self.<abstraction>)
modifies: {}

context List::prepend(Object o)
pre  : true
post : head(self.<abstraction>) = o and
       tail(self.<abstraction>) = self.<abstraction>@pre and
       validAbstraction(self, self.<abstraction>
modifies: self.<abstraction>
```

OCL

Please note that method `prepend` is a destructive method altering the list object. Therefore the modifies clause must contain the abstraction field. Any class implementing the interface `List` has to include at least all locations occurring in the valid abstraction formula ρ. In most cases the locations, which may change when the abstraction field changes will be specified explicitly. In order to realise this kind of mapping one can use the data groups provided by JML, which realise exactly such a mapping for model fields.

In order for practical use one usually requires that after the execution of a method the valid abstraction property is restored. a translation from a high-level specification to a JAVA CARD DL specification may add this postcondition explicitly, here it is stated explicitly in `prepend`'s postcondition.

The method contracts rendered by this specification can be used directly when verifying a program using one of the library classes resp. methods. Proving such a method contract sound is more difficult as the JAVA implementation cannot and does not know anything about the abstraction field itself and will therefore never 'change' it. The solution to be taken is that the specifier of an implementation must describe the necessary change by stating explicitly the

new value of the abstraction field. This can e.g. be done via an explicit `set` statement as used in JML for ghost fields.

Mapping OCL associations to JAVA CARD=DL Another application scenario is a proper mapping of OCL associations into JAVA CARD DL terms and formulas. When one uses OCL as specification language the model contains besides classes, attributes and operations also associations (aggregations, compositions) which have no direct representation on the program side. In order to interpret them one can introduce location functions, which model the associations as attributes, sets, lists etc. The chosen data type depends on the stereotypes restricting the kind of association.

5.2 Summary

This section contributed to the still open scientific problem how to close the gap between abstraction levels of data types. In particular, it analysis the situation where an abstract data type representation on the specification level has to be related to an imperative/object-oriented realisation on the program level for verification purposes. The presented suggestion expresses this relation using location functions bridging a concrete realisation of the data type with its abstract counterpart.

6 Modelling Java-Strings

6.1 The Java String Class

Java's String support differs in some aspects from the one found in other languages. For example, in contrast to the standard C or C++ solution a Java String is not identical to an array of characters. Following the object-oriented approach, they are realised as instances of a normal class part of the standard library, but benefit from some built-in support of the Java language.

For example, the operator `+` is overloaded to cover String concatenation. If one of the operator's arguments is a String, it becomes interpreted as String concatenation and a String conversion is performed if necessary. In presence of a conversion the operator `+` is not commutative, which is already described in [GJSB00, GJSB04].

The following list should cover most of the String specific properties:

String immutability. Strings are designed as an immutable class, i.e., after their creation it is not possible to change them. Immutability is a crucial property to allow some of the below mentioned features as literals of Strings, which behave similar to literals of primitive types. Immutability plays also a crucial role for the realisation of a String pool facility.

String Literals. A String literal in Java is a textual representation of the content enclosed in quotation marks. A String literal is an explicit reference to an instance of type `String` (with the specified content). Two equal literals refer always to the same String instance, so that for example the statement `System.out.println("abc"=="abc");` prints **`true`**. In order to mimic this literal behaviour Java provides a pool, which contains all String literals occurring in Java program and, in addition, all Strings that are part of a compile-time constant expression. A compile-time constant expression is an expression, whose value can be expressed as a literal and computed statically at compile-time. For example, the expression `"abc"+"def"` is a compile-time constant expression and evaluates to `"abcdef"`.

String Pool. The String pool is a technical detail mainly used to provide a sensible String literal support. Any two references which are part of the pool, reference objects with a different content.

The pool is set up at start-up (or more precise when a class is loaded) with references to String instances occurring in the class as literals or that are the result of constant compile time expressions.

Resolving a String literal means to test if the pool contains already an instance that represents the required content. If the test is positive the

literal is resolved to reference the already existing object. Otherwise a new String instance with the required content has to be created (or maybe, it has been already created to perform the test) and a reference to this object is added to the String pool.

The programmer may access the pool at runtime by invoking the method `intern` on a String object s. The invocation checks if an object equal to s is already a member of the pool and if not it will add the String s to the pool. Finally, it returns the reference to the now guaranteed to be existing pool member equal to s.

String conversion. The Java Language Specification(JLS) defines for any - primitive or reference - type, conversion rules that convert a member (instance) of the type to its canonical String representation. In case of an instance of a reference type, the corresponding `toString` method is invoked. In case of the null reference the conversion returns the text string `null` instead. The conversion rules for primitive types returns their natural textual representation.

There has been a minor flavour change between the second and third edition of the *Java Language Specification*. Influenced by the autoboxing and -unboxing feature in *Java 5*, the latter one specifies the conversion as the result of calling the `toString` method on the corresponding wrapper object. As an optimisation it is still allowed to implement the conversion in directly without creating intermediate objects, which is more in the flavour of *Java 2*.

String concatenation. Operator overloading for reference types is not supported by Java in general. The String concatenation operation, which uses the plus operator `+` is the exception of the rule. Java interprets the plus operator as String concatenation as soon as one of the operands is of type `String`. String conversions are performed if necessary.

With the approach introduced in this section, we do not aim to give a complete axiomatisation of Strings. The main concern is to allow to derive the most common properties about Strings in a user friendly way. The programmer using Strings will have to obey some rules if the program has to be verified. We believe that these rules should usually be followed by any well written program. The most important rule is:

Do not test for String equality via '==', use `equals` *instead.*

Some further problems are explicitly allowed optimisations to be performed by the compiler. These optimisations make it hard (impossible) to specify the String support in a platform independent way. This issue will be discussed in Section 6.3.

The followed line of attack is to model the content of a String separately from its concrete Java representation in terms of an abstract data type. When Strings are concerned one is usually interested in comparing a content to a

given literal. The abstraction to be presented is particularly adapted to this task since it abstracts away from object identity.

The remaining section assumes that all compile-time constant expressions have been already computed and replaced by the resulting literal.

6.2 Specification of the JAVA-String class

In this section we concentrate on modelling the JAVA-String class by mapping its operational semantics to operations of an abstract datatype. The specification makes use of location functions as defined in Def. 39.

6.2.1 The Abstract Data Type – AString

In this section we define the abstract datatype *AString* to be used to model the content of JAVA Strings. We keep this section short as the abstract data type does not differ significantly from the standard modelling found in many textbooks. An excerpt of the axiomatisation is shown in Fig. 6.1.

The axioms of the rewrite system above can be directly translated into the taclet language used in KeY. One problem to be solved is how to relate the abstract datatype with the concrete JAVA class `java.lang.String`.

For this purpose the location function

$$< \texttt{content} >: \texttt{java.lang.String} \rightarrow \texttt{AString}$$

is declared. It represents a 1-to-n relationship between Strings and their content. The plan is to axiomatise the function in a way resembling at invariants. In order to cope with the problem that a user can introduce an update that "hurts" these invariants in a way a valid JAVA program would not be able to, we have to restrict the application of the taclets to states which are reachable by a JAVA program. In our case we extend the predicate *inReachableState* to cover also these modelling. The justification for reusing this predicate is the tight integration of Strings into the JAVA standard.

Example 17. Assume a simple programming language where a statement consists of a command followed by a space and a list of additional arguments. A semicolon terminates the statement, e.g. `init 0 1 10;`. We specify the method `parseCommand` that returns the command part of a statement, i.e., the sequence of characters up to the first space:

```
\forall String s;(inReachableState & s!=null &
  s.<created>=TRUE -> {inStr := s}
     \<{ exc = null;
         try { cmdStr = parseCommand(inStr); }
         catch (Exception e) { exc = e; } }\> (
      \ifEx (int idx; s.<content>.getCharAt(idx, ' ')) \then (
         exc = null & inReachablestate &
         cmdStr.<content> =  subAString(s.<content>, 0, idx)
       \else (exc != null)))
```

```
constructors:  empty | cat: char -> AString -> AString

axioms:
 length: AString->int
 length(empty) = 0
 length(cat(c, lStr)) = 1+length(lStr)

 append: AString -> char -> AString
 append(empty, c) = c
 append(cat(fst, lStr), c) = cat(fst, append(lStr, c))

 concat: AString -> AString -> AString
 concat(lStr, empty) = lStr
 concat(lStr, cat(c, lStr')) =  concat(append(lStr, c), lStr')

 toAString: String -> AString
 toAString("") = empty
 toAString("a"+tail) = cat('a', toAString(tail))

 getCharAt: AString -> int -> char
 idx>0 & idx<length(lStr) -> getCharAt(cat(c, lStr), idx) =
                                  getCharAt(lStr, idx-1)
 getCharAt(cat(c, lStr), 0) = c

 subAString:AString -> int -> int -> AString

 subAstring cat(c, lStr) start end :=

   start >= 0 & end = start & end <= length(cat(c,lStr) ->
     subAString(cat(c, lStr), start, end) = empty

   start > 0 & end > start & end <= length(cat(c,lStr) ->
     subAString(cat(c, lStr), start, end) =
                      subAString(lStr, start-1, end-1)

   start = 0 & end > start & end < length(cat(c,lStr) ->
     subAString(cat(c, lStr), 0, end) =
                      cat(c, subAString(lStr, 0, end-1))
```

Figure 6.1: The abstract data type modelling strings.

6.2.2 String Conversions

As already described the Java Language Specification defines rules how to convert an instance of arbitrary type into a string. In this section we will by way of an example discuss the rule for the conversion of an `int` to its natural textual representation.

For example, in the following piece of code

JAVA
```
int a = 1;
String s = a + "23";
```
JAVA

the **int** typed variable `a` has to be converted into a String. In KeY following the symbolic execution paradigm, we need to make the conversion explicit. The rule initiating the conversion is:

KeY
```
convertIntToString {
  \find ( \<{.. #string1Loc = #int + #string2SE; ...}\> post)
  \varcond(\new ( #tmp, \typeof(#string1Loc) ))
  \replacewith ( \<{..
      java.lang.String #tmp = String.<convert>(#int);
      #string1Loc = #tmp + #string2SE; ...}\> post )
};
```
KeY

where `<convert>` is an implicit method defined besides others for **int**. There is no implementation for this implicit method, instead its behaviour has to be specified with taclet. This way it is also possible to easily adapt the specification to different implementations:

KeY
```
converterSpecForIntToString {
  \find ( \<{.. #stringLoc =
                java.util.String::<convert>(#int); ...}\> post)
  \replacewith ( {#stringLoc := convertIntToAString(#int)}
                                              \<{..  ...}\> post )
};
```
KeY

Note, that for the String conversion we do not allow any objects to be created, in particular no wrapper objects are used. Last, but not least, it remains to specify the AString converter function. In KeY a number literal is represented by a term of sort `Number`, a number is mapped to its canonical integer via the projection function Z. The encoding is in reverse order, i.e., the term `Z(3(2(1(#))))` represents the integer value 123.

KeY

```
\find ( convertIntToAString(Z(0(number)) )
\replacewith ( append( convertIntToAString(Z(number)), '0' ) )

....

\find ( convertIntToAString(Z(9(number)) )
\replacewith ( append( convertIntToAString(Z(number)), '9' ) )

\find ( convertIntToAString(Z(#)) )
\replacewith ( empty )
```

KeY

This kind of rule has to be specified for any primitive type. For reference types the conversion rule is even simpler. The reference `#ref` to be converted needs only to be replaced by the expression

```
#ref == null ? "null" :
   #ref.toString() == null ? "null" : #ref.toString()
```

After specifying the built-in conversions of Java. It proves useful to introduce a further conversion function intended to convert an array of characters into an AString by concatenating the arrays elements. This function is later used to specify one of the most common explicit constructors of class String. We need first an auxiliary function that maps the tail of an array after a given index `idx` into an AString.

```
charArrayToAStringHelper: char[] -> int -> AString
charArrayToAStringHelper array idx =
  if (idx>=0 & idx < array.length) then
    cat(array[idx],  charArrayToAStringHelper(array, idx + 1))
  else
    empty
 endif

 charArrayToAString: char[] -> AString
 charArrayToAString a = charArrayToAStringHelper a 0
```

As the function does not only depend on the value of its argument, but also on the component elements of the first argument, it has to be modelled as a non-rigid symbol. In order to improve the precision, it is modelled as a location dependent function.

6.2.3 String Creation

Implicit creation of Strings is by far more common than using an explicit constructor call. For example, at any time when an application of the concatenation operator is performed. According to the application interface specification of

the Java standard classes, the class String offers several constructors for the explicit creation of Strings from byte or character arrays with respect to a given encoding.

In order to model Strings accurately at the right abstraction level one has to decide how many implementation details have to be considered. Incorporation of too many internal details lead easily to strong restriction for a concrete implementation and reduces also the benefits of a higher abstraction level as one would have to deal for example with concrete character arrays storing the content of a String instead of the just using the AString abstraction.

Thanks to the immutability of Strings and that the Java Application Interface does not reveal too many details about the owned fields – in fact only one final static field is explicitly enumerated containing the lexicographical comparator for Strings, which needs not to be considered in this context – we can restrict ourself to a basic model consisting of the type itself and its declared operations. Nevertheless the constructor call will most likely create some character arrays, which requires to update static implicit fields in KeY that are used to model the instance creation process.

The necessity to know intermediately created but not escaping objects in order to gain a correct specification is a general model when specifying library functions with an unknown implementation. Forcing the specifier to explicitly mention all kinds of object to be created will put the programmer into narrow confines concerning the algorithm to use and the way to implement the algorithm. Further, it will be hard to update or exchange a (third party) library. On the other side simply allowing any object to be created will cause clumsy and lengthy updates, as quantification over types and attributes is not possible in KeY and the use of the assignable everything update operator – the anonymous update – erases nearly all benefits of representing assignable properties by updates.

Coming back to the specification of a String constructors and assuming that only character arrays are allowed to be created as intermediate non escaping objects during constructor's execution, the following specification can be used:

— KeY —

```
stringInstanceCreation {
 \find ( \<{.. #s = new java.lang.String(#se); ...}\>post )
 \replacewith (
   \if (#se == null) \then (
     \<{.. throw new java.lang.NullPointerException(); ...}\>post
   ) \else (
       {String.nextToCreate:=String.nextToCreate + 1,
        #s.<created>@(java.lang.Object):=TRUE,
        #s.<initialised>@(java.lang.Object):=TRUE,
        <content>(#s):=toAString(#se),
        char[].nextToCreate:=char[].nextToCreate + c}
           (inReachableState & \<{..  ...}\>post)))
 \add ( ==> c>=0, inReachableState ) };
```

— KeY —

where the constant `c` is newly introduced. Please note that the specification assumes that the implementation of the String constructor checks before the creation of any instance if the given argument is the null reference or not. The given specification ensures furthermore that the *inReachableState* property is preserved. Otherwise we would have to add further information about the created character arrays in order to use the specification in proofs. The function `toAString` chooses the conversion function according to the type and kind of the element matched by the schemavariable `#se`. The abstract data type function delegates the conversion to the matching conversion function, for example to `charArrayToAString` if the simple expression `#se` is matched against a character array.

6.2.4 String Literals

As already mentioned a String literal represents exactly one reference to an already created instance of type String at least in a Java reachable state. A problem when modelling String literals correctly is that they are a finite set, i.e., one has to forbid to introduce new String literals via cut-like rules. This means the only String literals that are allowed to be used in a proof are those that already exist in the context program explicitly or implicitly as part of a constant compile-time expression.

As consequence the following property is valid in any Java reachable state:

Theorem 5. *Let* Lit *denote the set of all literals occurring in the context program and* |Lit| *its cardinality. Then*

$$\mathcal{K}, s \models \mathit{inReachableState} \rightarrow \texttt{String.} < \texttt{nextToCreate} > \geq |\mathsf{Lit}|$$

holds for any state s.

As the set of literals is determined by the context program the exact value is known. The corresponding taclet is then

KeY

```
\assumes (inReachableState ==>)
\add( String.<nextToCreate> >= card(Lit) ==>)
```

KeY

where this taclet is either created dynamically or $card(Lit)$ is a meta operator returning the number of literals occurring in the context program.

Assigning a literal to a variable looks up in the pool for the reference to take. To implement the correct rule some additional work is necessary:

Definition 40 (Pool function)**.** The partially defined injective function *pool* maps an element of AString to an instance of type `java.lang.String`. It is defined for any AString element for which the natural String literal representative occurs in the context program.

Java reachable states have now to obey further restrictions according to the pool function, which is modelled as a non-rigid function symbol, which is left

unspecified for the undefined arguments, i.e., AString elements that have no representative in the context program. As long as the pool shall not be updateable by the programmer, i.e., no support for the method `intern`, the function can be declared as rigid.

The following additional property has to hold in any Java reachable state:

For any $l \in \mathsf{Lit}$ the reference returned by the pool function, that is $pool(l)$ has to be created and $pool(l).<\texttt{content}>$ equals the content described by the string literal l.

Notice that the above property already guarantees that *pool* is an injective function on Lit.

The String literal assignment rule can then simply be written as

KeY

```
stringLiteralAssignment {
  \find(\<{.. #s = #lit; ...}\> post)
  \replacewith({#s := pool(#lit^AString),
                  #s.<content> := #lit^AString}
                     \<{..  ...}\> post)
};
```

KeY

When assuming a normalised program in the sense that all constant compile-time expressions have been computed, then concatenation of String literals cannot occur as part of an expression inside a diamond. In case of a lazy normalisation implementation, the following rule will work

KeY

```
stringLiteralConcatenation {
  \find(\<{.. #s = #lit1 + #lit2; ...}\>post)
  \replacewith(\<{.. #s = #append(#lit1,#lit2);  ...}\>post)
};
```

KeY

where the `#append` is a program meta construct computing the concatenated literal and inserting it into the abstract syntax tree at the specified place.

6.3 String Pool and Optimisations

6.3.1 Complete Axiomatisation of the String Pool

In the above formalisation the String pool function has been realised as a rigid function. The chosen formalisation is possible as long as one does not intend to support the instance method `intern` of class `String`. For this support a solution is to define `pool` as a non-rigid location function. Additionally a new **`boolean`** typed instance field ¡interned¿ of type `java.lang.String` has to be introduced that indicates, if the instance is a member of the pool.

The price of the new flexibility comes along with a more complex definition of the Java reachable state predicate *inReachableState*. In a state where *inReachableState* is valid the following formulas must hold:

```
Let aString denote a variable of type AString
  pool(aString).<content> = aString

Let s denote a variable of type String then
  s.<interned> = TRUE ->
       (pool(s.<content>).<created>=TRUE & pool(s.<content>) = s)

for any lit ∈ Lit
  lit.<interned>=TRUE
```

A valid taclet matching the semantics of an `intern()` invocation is then

KeY

```
\find(\<{.. #v = #s::intern()@(java.lang.String); ...}>post)
\replacewith( \if (#s.<interned>=TRUE) \then
                 ( \<{.. #v = #s; ...}\>post )
              \else
                 (  {s.<interned>:=TRUE, pool(s.<content>):=s}
                       \<{.. ...}\>post)
)
```

KeY

6.3.2 Further JLS conform Optimisations

The Java Language Specification allows a number of further optimisations to be performed by the compiler of the virtual machine. For example, a virtual machine may create intermediate `StringBuffer` instances when evaluating a chain of String concatenations. The problem is that then the concatenation operators implicit assignable clause must contain also the implicit fields of type `StringBuffer`. Optimisations of similar kind can be also observed for String conversions.

In our approach we have not considered the intermediate creation of StringBuffer instances. The problem is not that this would be hard to realise its more that the JLS also allows similar optimisations without specifying them further. The question is which proofs can be believed to be platform independent with respect to optimisations. Currently the cause of the problem seems to be that JAVA CARD DL allows the inspections of properties which are not accessible from the Java language itself.

6.4 API specification challenges

Some of the above mentioned obstacles when formalising Strings reoccur in different setting, for example when writing a specification for library methods.

While the library implementation may change, the specification usually shall remain unaltered.

In the next few paragraphs, we will shortly describe some of the problems in more detail.

6.4.1 Assignable Clause

Assignable clauses capture change information. Their usage allows to preserve as much knowledge about the prestate of a method as possible and to erase only those parts, which may have become invalid after the method call returned. The drawback are restrictions in the choice of (realisations of) algorithms to implement the intended functionality.

The consequences can be moderated when the fields that are changed are only private fields and are also only required in the specification of private invariants and methods. As it is possible to reduce the logic model of the library to non-private fields and methods. This also encourages a programming style that prefers fields to be declared private and enforces data encapsulation. If the program fulfils further encapsulation properties, the implementation may also use package private or protected fields.

Another problem is caused by the current formalisation of object initialisation in KeY. The used formalisation unveils more details about the used memory model than done by the JLS. It is for example possible to query the order in which two objects of the same dynamic type have been created by comparing their indices. As described in Sect. 2.2.3 the index of the object to be taken next is stored in a static field implicitly declared in each class. In principle the assignable clause has to contain all implicit fields with a connection to initialisation or object creation of all objects and types. The current implementation requires to specify the assignable clause in detail i.e., by enumerating all implicit fields to be changed or to use the anonymous update erasing any knowledge about the prestate.

In the subsequent paragraphs two and a half solutions are presented to cope with this problem.

Restricted type quantification in assignable clauses In order to address static fields of the same name and type, but declared in different types or by quantifying over all objects of certain types in assignable clauses, one solution is to add special quantifier expression for assignable clauses to the specification language.

For example, the both informally introduced quantifiers are sufficient for the above mentioned object creation purposes:

```
   \forallTypes T;TypeRestriction(T);T.attr@(T)
or
   \forallTypes o:T;TypeRestriction(T);o.attr
```

where `TypeRestriction` is a decidable Boolean expression, which can be evaluated statically. A valid expression can be composed according to the following grammar:

```
TypeRestriction   ::=
     PackageRestricted | InheritanceRestricted |
     TypeRestriction ( '&' | '|' ) TypeRestriction |
                    '!' TypeRestriction | true
PackageRestricted ::=
     'inPackage'  '(' PackageIdentifier ')' |
     'inPackages' '(' PackageIdentifierWildCard ')'

InheritanceRestricted ::=
      TypeName ('=' | '<' | '<=' | '!=') TypeName

PackageIdentifier ::= IDENT '.' IDENT
PackageIdentifierWildCard ::= PackageIdentifier ('.' '*')?
```

Note 22. These quantification expressions evaluate to sets of attributes. Please note that the sets are always finite as the set of all types is finite in our logic.

Example 18. The location descriptor expression

```
\forallTypes T;inPackages('java.*');T.<nextToCreate>@(T)
```

describes the set of all attributes named `<nextToCreate>` of all class types that are members of the standard library package `java` or one of its subpackages.

There are two different approaches how to make use of these extended location descriptor sets.

The first solution is to leave the logic unchanged and to eliminate these quantifiers when translating the location descriptor to an anonymous update. The drawback is that the occurring updates will often become clumsy.

Example 19. Assume that the assignable clause of a method `m` is given as

```
@assignable
   \forallTypes T;inPackages('java.*');T.<nextToCreate>,
        \forallTypes o:T;inPackages('java.*');o.<created>
```

In words: The given assignable clause allows the implementation of method `m` to create an arbitrary number of objects as long as the dynamic type of the created objects is a member of the `java` package or any of its subpackages. As the Boolean expression used for the type restriction is by definition statically decidable and by remembering Note 22, the assignable clause can be translated to an anonymous update as follows:

```
{ java.lang.Object.<nextToCreate>  := c1, ...,
  java.lang.Integer.<nextToCreate> := cm,
  \forall java.lang.Object o;
       \if (java.lang.Object::exactInstance(o)=TRUE | .. |
            java.lang.Integer::exactInstance(o)=TRUE)
            o.<created>:=d(o) }
```

In order to avoid clumsy updates resulting from the translation, it is also possible to extend the language of quantified updates by these new quantifier expressions. As the quantification is statically decidable, the additional update rules can be of a very simple kind which mainly translate the new quantifications lazily when an update is applied.

One problem that remains is that the values of the newly created attributes may have been changed. Assume a void method which only creates an Object of type `T` and forgets it afterwards. Even for such a simple case the formula

```
\forall T o; o.a = 1 -> \<{ m(); }\> \forall T o; o.a = 0
```

is not valid. The implications of this fact are rather huge as one now has to either anonymise all fields by enumerating them or to extend the language for quantified updates once again. In general the first suggestion will lead to really long updates, although the updates can be kept small in presence of a reduced logic model as mentioned above, simply spoken only the types and methods of the library class are represented and used.

We treat the second suggestion in more detail as it will be reused by the approach to be presented in the next paragraph. Anonymising all attributes that belong to an object can be done fairly simple by introducing a new kind of assignment pair. Its syntax is $*(l) := c\_*$, where l denotes a term and c_* a nullary function symbol of sort $\top$. Given a Kripke Structure $\mathcal{K}$, a state S and a variable assignment β such an assignment pair evaluates to a set of location triples

$$\langle f, (val_{\mathcal{K},S,\beta}(l)), I(cast(typeof(f)))(c\_*(val_{\mathcal{K},S,\beta}(l)))\rangle$$

The update rule for an application on an attribute access term $o.a$ of type T is then

$$\{*(o) := c\_*\}\ o.a \rightsquigarrow (T)c_*(o)$$

Alternative Formalisation of Object Creation The second solution changes the way object creation is formalised in KeY. Assume the existence of a logic type `FinObjectSet` together with a set of functions and axioms. The type shall be axiomatised as a generated finite set (i.e., inclusive a structural induction rule) and the typical constructors, functions and queries for operations on sets. Further, only elements of type `java.lang.Object` are allowed to be elements of this kind of set.

Instead of the implicit attributes `<nextToCreate>` and `<created>` used to keep book of created objects, the new nullary location function `createdObjects` of type `FinObjectSet` is used. In order to query if an object is created, the location dependent Boolean valued function `isCreated[createdObjects;]` is used instead of an implicit Boolean attribute like `<created>`. The formal definition of this function given as taclet is:

— KeY —

```
\find (isCreated[createdObjects;](o))
\replacewith (contains(createdObjects, o))
```

— KeY —

At any time when a new object is created, the set is updated to contain the newly created object. Specified in taclet code this looks like

KeY

```
\find(\<{.. #newObject = T.<getObject>(); ...}\>phi)
\replacewith( {createdObjects:=add(createdObjects,
                  T::<get>(maxIndex(createdObjects)+1)),
 #newObject:=T::<get>(maxIndex(createdObjects)+1)}
                                        \<{.. ...}\>phi)
```

KeY

The index of the newly created object is determined by querying for the maximal index in the set of already created objects and increasing the index by one. This formalisation allows us also to derive that the newly created object cannot be an element of `createdObjects`. This formalisation ensures also that object creation remains deterministic.

The new formalisation simplifies also the definition of *inReachableState* predicate as it has not to take care if the `<created>` attribute is consistent with the attribute `<nextToCreate>` or if the field `<nextToCreate>` is in the allowed range, i.e., non-negative. It is also possible to exclude if a state can be reached from another one by looking at the subset relation ship as objects are never deleted.

The main improvement has been also our motivation, that is to allow a concise and comfortable specification of methods regarding their assignable clause. It is now possible to change the semantics of a methods assignable clause to allow the creation of an arbitrary number of objects.

When applying the method contract rule (see Sect. 2.2.6) it will update the set of `createdObject` by adding a finite (but unknown) number of new objects. The update looks like

```
{createdObjects:=union(createdObjects, anonCreatedObjects),
 \for java.lang.Object o; *(o):=c_*}
```

Note 23. The used constant `c_*` has to be new at each application of the method contract rule.

Note 24. In addition the method contract rule application has to add the following axioms to the antecedent:

- the finite sets `createdObjects` and `anonCreatedObjects` have no element in common, i.e., their intersection is empty
- the indices of the objects in `anonCreatedObjects` are greater than the maximal object index to be found in `createdObjects` or more precisely:

```
maxIndex(anonCreatedObjects) -
   minIndex(anonCreatedObjects) = size(anonCreatedObjects) &
minIndex(anonCreatedObjects) = maxIndex(createdObjects) + 1
```

In order to be sure that the new formalisation is an improvement in contrast to the current one, it needs to be taken care that it is easily possible to prove that two (or more) successively created objects are not the same object. The current formalisation reduces this question to a simple arithmetic problem, namely the comparison of the object indices, i.e., to prove that

```
T::<get>(i) = T::<get>(j)
```

can be reduced to a prove of `i=j`. This reduction is possible in the new formalisation as well and can be used as efficiently as before since the object creation chooses the index to be taken deterministically.

6.4.2 Abstraction Level

An abstraction layer as for Strings may also be of advantage for other data types than Strings. For example, list or stream implementations could be related to their canonical abstract data type realisation. The procedure would be similar to Strings. These abstractions are suitable if one is mainly interested in the content of a list and not its real implementation.

In order to relate an abstract data type model to a concrete implementation the following items need to be done:

- Introduction of a location function to relate the concrete type with its abstraction, as seen for Strings.
- A relation defining non-rigid predicate, which evaluates to true if the real data structure and its abstraction are consistent. For Strings we have not introduced a separate predicate, but reused and redefined the *inReachableState* predicate.
- Specify the methods of the classes and interfaces in terms of the newly introduced data type.

6.5 Summary

This section analysed and presented a solution for a faihtful modelling of Java Strings on the program logic level. It provides an abstraction that allows for convenient specification and verification in the presence of character strings.

Part III

Case Study

7 Case Study: The Schorr-Waite Algorithm

7.1 Motivation

The Schorr-Waite graph marking algorithm ([SW67]) named after its inventors Herbert Schorr and William M. Waite has become an unofficial benchmark for the verification of programs dealing with linked data structures.

It has been originally designed with a LISP garbage collector as application field in mind and thus, its main characteristic is low additional memory consumption. The original design claimed only two markers per data object and, more important, only three auxiliary pointers at all during the algorithm's runtime. It is the latter point, where most other graph marking algorithms lose against Schorr-Waite and need to allocate (often implicitly as part of the method stack) additional memory linear in the number of nodes in the worst case. These resources are used to log the taken path for later backtracking when a circle is detected or a sink reached.

Schorr and Waite's trick is to keep track of the path by reversing traversed edges offset by one and restoring them afterwards in the backtracking phase of the algorithm. A detailed description including the Java implementation to be verified is given in Section 7.2.

Formal treatment of Schorr-Waite is challenging as reachability issues are involved. Transitive closure resp. reachability is beyond pure first order logic and some extra effort has to be spent to deal with this kind of problems. On the other side, the algorithm is small and simple enough to serve as a testbed for different approaches. The specification is described in detail in Sect. 7.3, some notes on the actual verification can be found in Sect. 7.4

7.2 The Graph Marking Algorithm "Schorr-Waite"

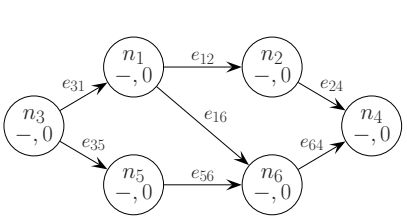
7.2.1 Description

As usual a *directed graph* G is defined as a set of vertexes V and edges $E \subseteq V \times V$. The directed edge $s \rightharpoonup t \in E$ connects source node $s \in V$ with target node $t \in V$, but not vice versa. We call node t a direct *successor* of node s (resp. s a direct *predecessor* of t).

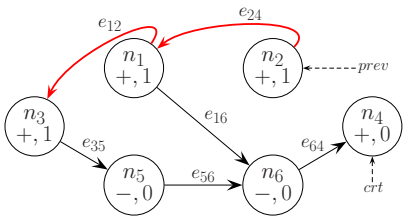
For sake of simplicity, we require that each edge e is labelled with a unique natural number $l(e)$ where $l : E \to \mathbb{N}$. The labelling allows us to put an order on all outgoing edges $e_i := s \rightharpoonup_i t_i, i \in \{1, \ldots, n\}$ of a node s, which complies with the natural number ordering $\leq$ of the corresponding labels $l(e_i)$.

When speaking of visiting all children (of a node s) from left-to-right, we mean in fact that all direct successors of s are accessed via its outgoing edges in ascending order of their labels. We refer to the target node of the edge with the i-th smallest label of all outgoing edges of node s as the node's i-th child.

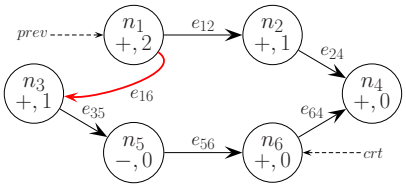
In addition each node is augmented with a flag `visited` and a field `usedEdge`, which will be used to store the number of the most recently visited child via this node (or equivalently the corresponding edge label).



(a) Initial unmarked Graph



(b) Visiting node n_4 via n_2; edge e_{12} (e_{24}) reverses formerly traversed e_{31} (resp. e_{12})



(c) Visiting node n_6 via n_2; backtracking restored the formerly modified edges e_{12}, e_{24}

Figure 7.1: Illustration of a Schorr-Waite run: edges coloured red (curved) have been modified to encode the taken path; pointers *crt*, *prev* refer to the currently resp. previously visited node

In the subsequent four additional pointers are required:

- `current` and `previous`, whose intended purpose is to refer to the currently respective previously visited node and
- the two helpers `next` and `old`[1].

Given a directed graph G as for example shown in Fig. 7.1 and a designated node s, Schorr-Waite explores G starting at node s applying a left depth-first strategy:

1. outgoing from the currently visited node `current` the leftmost not yet visited child `next` is selected and the taken edge e redirected to target the node referenced by pointer `previous`. The `usedEdge` field of `current` is used to keep the label $l(e)$ of the reversed edge in order to restore

[1] in fact one of the helper variables is superfluous as we could reuse the other. The drawback would be a more generic and unintuitive naming like `tmp`.

edge e later in the backtracking phase (step 2). Afterwards `previous` is altered to point to our current node, while pointer `current` moves onto node `next`. Finally, the new `current` node becomes marked as visited. Continue with step 1.

2. if all children of the node referred to by `current` have already been visited or it is a childless node and `current` $\neq s$, then a backward step is performed. Therefore the edge via which `current` has been accessed and remembered in the `usedEdge` field of node `previous` during step 1, is restored this means rebend to its original target `current`, but not before rescuing its current target using pointer `old`. Now pointer `current` can be reset to the node referenced by `previous` and – last but not least – `previous` is moved back to node `old`. Continue with step 1.

After all reachable nodes have been visited the algorithm terminates when after a backtracking step the starting node s is reached. At this time the original graph structure has been also restored.

7.2.2 Implementation

The design of our Java implementation to be verified is illustrated in Fig. 7.2. The graph nodes are modelled as instances of class `HeapObject`, where each instance contains a `children` array, whose i-th component contains the node's i-th child.

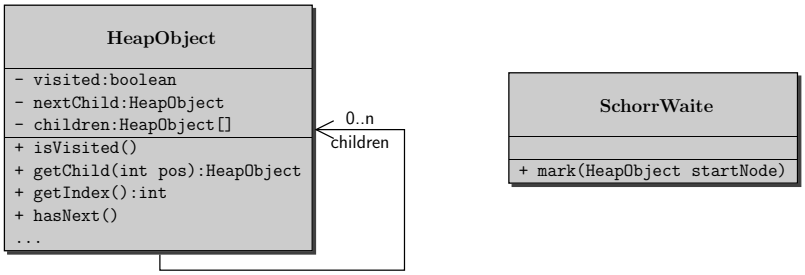


Figure 7.2: Class diagram showing the involved participants

All `HeapObject` instances provide a rudimentary iterate facility to access their children. Therefore they implement an integer index field, which contains the array index of the child to be visited next. Method `hasNext` tests if the index field has reached the end of the array and therewith all children of the node have been accessed. The index field is used to realise the `usedEdge` field of the description above. In fact, `usedEdge` is equal to the value stored in the index field minus one.

The JAVA implementation of the algorithm itself is realised as method `mark` of class `Schorr-Waite` shown in Fig. 7.3. Invoking `mark` with a non-null start node handed over as argument starts the graph traversal. The method assumes that before it is invoked, all `HeapObject` instances have no marks set.

```
public void mark(HeapObject startNode) {
  previous = null;
  current  = start;
  current.setMark(true);
5 while (start != current || start.hasNext()) {
    final HeapObject tmpChild;
    if (current.hasNext()) {
      final int nextChild = current.getIndex();
      tmpChild = current.getChild(nextChild);
10    if (!tmpChild.isMarked()) {// forward scan
        current.setChild(nextChild, previous);
        previous = current;
        current.incIndex();
        current = tmpChild;
15      current.setMark(true);
      } else {
        // already visited or no child at this slot
        // proceed to next child
        current.incIndex();
20    }
    } else {
      // backward
      final int ref2restore = previous.getIndex() - 1;
      tmpChild = previous.getChild(ref2restore);
25    previous.setChild(ref2restore, current);
      current  = previous;
      previous = tmpChild;
    }
  }
30}
```

Figure 7.3: These few lines implement the core of Schorr-Waite

The first lines of method `mark` initialise the fields `current` and `previous`. The starting node s of the previous section is handed over as the method's argument referred to by parameter `startNode`. It becomes also the first node `current` points to. All other pointers are set to **`null`** for the first. Before the while loop is entered, the starting node `startNode` is marked.

After these preparations the graph is traversed in left depth-first order as long as `current` has not yet returned to the starting node `startNode` or if there are still children of node `startNode` to be checked.

If node `current` has a child left, a forward step is performed (lines 7-21):

line 8 the `nextChild`th component of node `current`'s children array (that is the `current`'s next not already accessed child) is assigned to variable `next`

lines 10-15 these lines are entered in case that the `HeapObject` instance referred to by `next` has not already been visited, which is tested by calling method `isMarked` in the conditionals guard. First the taken edge, i.e., the `nextChild`th component of `current`'s children array, has to be redirected to the node variable `previous` refers to (line 11). In the succeeding lines, field `nextChild` of node `current` is updated, pointer `previous` is moved toward the node `current` refers to, and finally, node `next` becomes the new `current` node. At the end the new `current` node is marked as visited with help of method `setMark`.

line 19 is only executed in case that node `next` has been already marked in a previous step

Whereas lines 22-28 are executed if node `current` has no remaining children to be visited. In this case a backward step has to be performed:

line 24 the `nextChild-1`th child of node `previous`, which stores the penultimate node is memorised in `old`

line 25 the `nextChild-1`th children array component of node `previous` is restored, i.e., redirected to node `current`

lines 26-27 finally, `current` becomes `previous` and `previous` is set to the node stored in `old`.

7.3 Specification

7.3.1 Proof Obligation

As already mentioned the properties of interest are that the original graph structure is preserved after the algorithm terminates and of course, that all reachable nodes have been visited.

These properties can be formalised in JAVA CARD DL as shown in Fig. 7.4, which is further explained in the following: The lines 1 to 17 formalise properties of instances of class `HeapObject` derived from decisions made when designing that piece of software like

line 1 any `HeapObject` has a non-null reference to a children array. Consequently sinks are modelled as nodes with a zero length array.

lines 3-4 the length of the children arrays in non-negative (this property is necessary as it is not guaranteed by the logic definition itself).

lines 6-7 no two `HeapObject`s share the same children array.

lines 9-11 the `nextChild` counter of an `HeapObject` is in range, i.e., its value is between and including zero and the length of its children array.

lines 13-17 the entries of the children array are not null.

KeY

```
  \forall HeapObject ho; (!ho = null -> !ho.children = null)
&
  \forall HeapObject ho;
     (!ho = null -> ho.children.length >= 0)
&
  \forall HeapObject h1;\forall HeapObject h2;
      (!h1 = h2 -> !h1.children = h2.children)
&
  \forall HeapObject ho; (!ho = null ->
     (ho.nextChild >= 0 &
      ho.nextChild <= ho.children.length))
&
  \forall HeapObject ho;
     \forall int i;(!ho = null &
        0 <= i & i <  ho.children.length ->
                              !ho.children[i] = null)

& inReachableState
& !sw = null
& !startNode = null
& sw.<created> = TRUE
& startNode.<created> = TRUE
& \forall HeapObject ho;
     (!ho = null -> ho.visited = FALSE & ho.nextChild = 0)
& \forall HeapObject ho;\forall int i;
     ho.children[i] = childrenPre(ho, i)
->
  \[{
      sw.mark(startNode);
   }\] \forall HeapObject x;\forall int n;(
          !x = null & reach[...](startNode, x, n) ->
             (x.visited = TRUE &
              \forall int i;(i>=0 & i<x.children.length ->
                          x.children[i] = childrenPre(x,i))))
}
```

KeY

Figure 7.4: JavaCardDL specification for structure preserving, correct and complete graph coverage of Schorr-Waite

In high-level specification languages these properties are usually specified as (instance) invariants. Besides these properties the following propositions are required to hold

line 18 the method is called in a state actually reachable by a JAVA program

lines 19-22 the instance on which the graph marking algorithm is called is not null and created. Same holds for the node given as starting node.

lines 23-24 initially all instances must not be marked as visited and their `nextChild` counter is set to 0. In high-level specification languages this property would be expressed as preconditions, the other properties in this list would be translated when compiling the high-level specification language to a low level one like JAVA CARD DL.

lines 25-26 this properties allows to refer to the old graph structure in the post state by providing a 'copy' in terms of a rigid function definition.

such that when the graph marking algorithm terminates in its final state all reachable notes have been visited (line 32) and the graph structure is the same as before the method call (lines 33-34).

7.3.2 Encoding the Backtracking Path

For the specification of the loop invariant it turns out to be useful to define a relation *onPath*, which describes the backtracking path. We formalise the relation as the characteristic function of the set of nodes lying on the backtracking path by means of an auxiliary non-rigid predicate with explicit dependencies $\texttt{onPath}[*.\texttt{children}[*]; *.\texttt{nextChild}] : HeapObject \times HeapObject \times int$.

For sake of shortness and readability, we will skip the accessor list for the *onPath* predicate from now on. Formally, the non-rigid predicate *onPath* is defined as follows:

Let σ denote a state, x, y terms of type `HeapObject` and n an integer term then

$$\sigma \models onPath[\ldots](x, y, n)$$
$$\text{iff.}$$
$$n >= 0 \text{ and there exist terms } x = u_0, \ldots, u_n = y, \text{ such that}$$
$$\text{for all } 0 \leq i < n :$$
$$\sigma \models u_{i+1} \doteq u_i.children[u_i.nextChild - 1]$$
$$and$$
$$\sigma \models n > 0 \; ->$$
$$!\, u_i \doteq null \;\&\; u_i.nextChild > 0 \;\&\; u_i.nextChild <= u_i.children.length$$

Notice that $onPath[\ldots](x, y, 0)$ is equivalent to $x \doteq y$. The semantical definition of *onPath* is reflected by the calculus in form of a recursive definition:

KeY

```
onPathDefinition {
  \find(onPath[...](t1, t2, n))
  \replacewith(n >= 0 &
      ((t1 = t2 & n = 0) |(t1 != null & t1.nextChild > 0 &
            t1.nextChild <= t1.children.length) &
          onPath[...](t1.children[t1.nextChild - 1], t2, n-1))))
};
```

KeY

The recursion is well founded and required to formalise the existential statement given in the semantical definition. For convenience reasons, we use additional taclets which can be derived directly from the rule onPathDefinition:

KeY

```
onPathBase {
  \find(onPath[...](t1, t2, 0))
  \replacewith(t1 = t2)
  \heuristics(simplify)
};

onPathNull {
  \find(onPath[...](null, t2, n))
  \replacewith(n = 0 & t2 = null)
  \heuristics(simplify)
};

onPathEffectlessUpdate {
  \find(onPath[...](t1, t2, n)) \sameUpdateLevel
  \varcond(\notFreeIn(hov,loc),\notFreeIn(iv,n,t1,loc))
  \replacewith({loc.children[locIdx]:=val}onPath[...](t1,t2,n));
  \add(==> \forall iv;(iv>=0 & iv<=n -> !onPath[...](t1,loc,n)));
  \add(==> \forall hov;(hov.children=loc.children -> hov=loc))
};

onPathEffectlessUpdate2 {
  \find(onPath[...](t1, t2, n)) \sameUpdateLevel
  \varcond(\notFreeIn(iv,n,t1,loc))
  \replacewith
     ({loc.nextChild:=val}onPath[...](t1, t2, n))
  \add(\forall iv;(iv>=0 & iv<=n ->
                   !onPath[...](t1, loc, n)) ==>);
  \add(==> \forall iv;(iv>=0 & iv<=n ->
                          !onPath[...](t1, loc, n)))
};
```

```
onPathNoCycle2 {
  \find(onPath[...](t1, t1, n2) ==>)
  \varcond(\notFreeIn(i, t1))
  \add((t1 = null & n2 = 0) | n2 = 0 ==>);
  \add(==> \exists i; onPath[...](t1, null, i))
};

onPathTransitive2 {
  \find(onPath[...](t1, t2, n2) ==>)
  \varcond(\notFreeIn(i, t1, t2, t3, n2))
  \add(\forall i;(onPath[...](t2, t3, i)
           -> onPath[...](t1, t3, n2 + i)) ==>)
};
```

KeY

With this work done, we can now express the property that a node `x` is on the backtracking path by:

$$\backslash\text{exists } int\ n\ onPath[\ldots](previous, x, n);\ \mid\ \texttt{x} = \texttt{current}$$

where `previous` and `current` are the reference variables declared in method `mark`. Note, that we have included the `current` node to belong to the backtracking path.

7.3.3 The Loop Invariant

The most critical part of the specification is the loop invariant as most of the later verification depends on a sufficiently strong invariant. The **`while`** invariant rule used in KeY takes change information into consideration and allows to reduce the complexity of the invariant.

The loop's assignable set is

```
{sw.current, sw.previous,
 \for HeapObject h;h.children[*],
 \for HeapObject h;h.visited,
 \for HeapObject h;h.nextChild}
```

In the first line all possibly altered method-local pointers are enumerated. The remaining lines denote all the fields of nodes that are likely to be changed. The assignable set is a conservative approximation in principal it would be sufficient to restrict to fields of nodes reachable from the starting node. Please note that the local variable `startNode` is not part of the assignable clause and remains therewith unchanged during loop execution.

The complete loop invariant is shown in Fig. 7.5 and in Fig. 7.6. Some useful explanations:

lines 1-5 this part of the loop invariant records properties to be satisfied by the auxiliary fields `previous` and `current` resp. the nodes they refer to:

- the node `current` refers to is marked as `visited`.

KeY

```
  sw.current.visited=TRUE & sw.current!=null
&
  (sw.current=startNode <-> sw.previous=null)
&
  startNode.visited=TRUE
&
  \forall HeapObject ho;\forall int i;(
    (ho!=null & i>=0 & i<ho.children.length) ->
      (ho.children[i]=null <->
       (ho=startNode & i=startNode.nextChild-1 &
        startNode!=sw.current)))
&
  \forall HeapObject ho;
     (ho!=null -> (ho.nextChild <= ho.children.length &
                   ho.nextChild >= 0))
&
  \exists int end;(onPath[...](sw.previous, null, end))
&
  !\exists int i;(onPath[...](sw.previous, sw.current, i))
&
  \forall HeapObject ho;(ho!=null & ho.visited=FALSE ->
     (ho.nextChild=0 &
       \forall int i;((i>=0 & i<ho.children.length) ->
                      ho.children[i]=childrenPre(ho,i))))
& ... continued in Fig. 7.6
```

KeY

Figure 7.5: Loop invariant (part I)

- the `current` field keeps never the **`null`** reference. This fact is, for example, useful to derive the absence of `NullPointerException`s.
- the `previous` field is usually not **`null`** except when `current` refers to `startNode`.

In addition one has to record that the `startNode` has already been visited. This is necessary as the assignable clause is only a safe approximation containing the visited field of all nodes.

lines 7-11 as mentioned before the `children` array of a node contains never **`null`**. This property is temporarily violated, when leaving the start node. In that situation (see item above) `previous` is **`null`**, which is the value the `nextChild-1` child entry of the `startNode` will be set to. Please note that when leaving the loop, `current` refers to the same node as `startNode` and therewith the class invariant of `HeapObject` will have been restored.

lines 13-15 state that the value of the `nextChild` field of a node is always between and including 0 and `children.length`.

lines 17-19 these lines rescue knowledge about the backtracking path, namely that the backtracking path is acyclic and terminated by **`null`**. Further `current` does not lie on the backtracking path.

lines 24 this part of the loop invariant expresses intuitively that the algorithm cannot have changed any node or sub-graph, which has not yet been reached.

The loop invariant continues with Fig. 7.6. While the previous part concentrated mainly on some general properties and rescued knowledge about not (yet) visited parts of the graph, the remaining loop invariant focuses on visited nodes:

lines 2-4 for a visited node which is not the actual `current` node the range of `nextChild` is strictly greater than zero except if the node is a sink (i.e., no outgoing edges).

lines 5-8 the `children` entries of a visited node above and with the `nextChild` entry have not yet been looked at by the algorithm and are therewith unchanged. This information is necessary to establish that the loop invariant is preserved, but not needed in the use case of the loop invariant. In the latter case this part of the conjunction is trivially true as the antecedent of the implication evaluates obviously to false.

lines 10-16 this part of the invariant captures the old value of the `nextChild-1` entry of a node on the backtracking path, which has been used to encode the backtracking path itself. The identity used is shown in Fig. 7.7 and valid for all nodes on the backtracking path except for the node `previous` points to. Further all nodes on the backtracking path are visited and have a `nextChild` index greater then zero (attention: as the `current` node is no member of the backtracking path, a sink will also never lie on the backtracking path).

KeY

```
... for the first part of the invariant see Fig. 7.5
& \forall HeapObject ho; ( ho!= null & ho.visited=TRUE &
     ho!=sw.current ->
        (ho.nextChild>=1 | ho.children.length=0))
& \forall HeapObject ho;(ho!=null & ho.visited=TRUE ->
     (\forall int i; (
         (i>=0 & i>=ho.nextChild & i<ho.children.length) ->
            ho.children[i]=childrenPre(ho, i))))
&
  \forall HeapObject ho;((ho!=null &
     \exists int i; onPath[...](sw.previous, ho, i)) ->
        ((childrenPre(ho.children[ho.nextChild-1],
                      ho.children
                           [ho.nextChild-1].nextChild-1) = ho
         | (ho.children[ho.nextChild-1]=null & ho=startNode))
        & ho.visited=TRUE & ho.nextChild>=1))
&
  (sw.previous != null -> childrenPre(sw.previous,
                  sw.previous.nextChild-1) = sw.current)
&
  \forall HeapObject h;((h!=null & h.visited=TRUE) ->
     (\forall int i; ((i>=0 & i<h.nextChild-1) ->
         (h.children[i].visited=TRUE &
          h.children[i]=childrenPre(h,i)))
      &
      ((!(\exists int dist;
             onPath[...](sw.previous, h, dist)))
        -> ((h!=sw.current -> h.nextChild=h.children.length) &
             ((h.nextChild>0) ->
               (h.children[h.nextChild-1].visited=TRUE   &
                h.children[h.nextChild-1]=
                             childrenPre(h,h.nextChild-1))))
      )))
&
  (startNode=sw.current ->
     (\forall int i;((i>=0 & i<startNode.children.length) ->
         startNode.children[i]=childrenPre(startNode,i)) &
      (startNode.nextChild>0 ->
         startNode.children
                      [startNode.nextChild-1].visited=TRUE)))
```

KeY

Figure 7.6: Loop invariant (continued)

lines 18-19 These lines fix the hole of the previous item for the `previous` node.

lines 21-33 For any visited node all entries of the children array from zero to (and including) `nextChild-2` have been restored and these children are of course marked as visited. Except for nodes on the backtracking path this is also valid for the `nextChild-1` child of the corresponding node.

lines 35-40 the above item does not cover the case where `current` refers to the `startNode`. The hole is patched here.

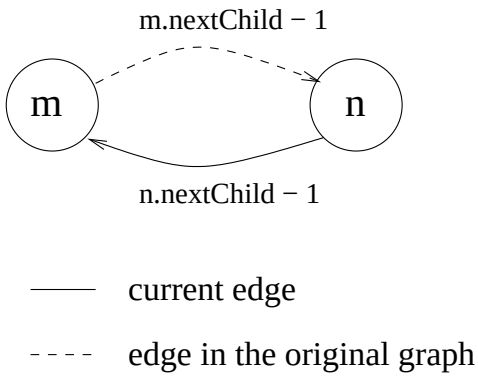


Figure 7.7: Identity used to reverse the backtracking path.

7.4 Verification

The proof that all reachable nodes have been marked as visited and that the graph structure has been restored after the method terminates consists of 17936 nodes distributed on 392 branches. To complete the proof 1017 interactive steps had been necessary. The relative high number of interaction is mainly due to

- manual instantiation of quantifiers: The KeY-version used for this proof did not yet support automatic instantiation of quantifiers. The built-in support for calling external decision procedures like *Simplify* had not been used.
- taclets dealing with reachability issues need mainly to be performed manually as for example no automatic strategies have been implemented yet to exploit transitivity of reachable automatically or that updates do not influence the truth value of a non-rigid predicates in a sensible manner.
- control of the proof structure. Many of the interactions would not have been necessary in principle, but have been useful to keep track of the current part of the property to be proven.

Reducing the number of necessary interaction is planned for the near future. It will be interesting to see how many interactions are necessary when using the recent version of KeY, which supports automatic instantiation of quantifiers.

Some information about the verification itself. After some initial steps the proof splits up into the three branches created by the loop invariant rule.

The first branch requires to show that loop invariant is initially valid. The branch can be closed straight forward, as the simple parts of the loop invariant like that the current field is not null are established by the statements before the loop is entered. The more difficult parts of the loop invariant collapse immediately as the initially no field is marked visited, the backtracking path is empty, etc.

The branch where the preservation of the invariant has to be proven is the most difficult and lengthy one. The loop condition forces a proof split into two cases:

1. the current node is not equal to the start node and therefore at least further backward steps have to be performed.

2. the start and current node are the same, but not all of its children have been treated yet.

For each of the two cases the proof tree splits up into three further sub-proofs. The proof splitting is induced by the program structure. The loop body splits the control flow into separate flows.

The first flow is entered in case of a forward step, which itself causes a further split depending if the node chosen for the forward step has already been visited or not. The second one implements the backward step logic.

This multiplies to six branches at all. Only the backward step branch in case that the current and start node are the same can be closed within a few steps as in this case no backward step is possible. For any of the other five branches one has to show independently that the loop invariant is preserved.

The only remaining open proof branch is the "use case" of the loop invariant rule. In order to close this branch one has to show that for any node `ho` reachable from `startNode` the following holds: the visited flag is set and the children of `ho` are the same as before. The induction is performed over the path length from the start node to `ho`. Fig. 7.8 shows the induction hypothesis: the induction variable is `iv`, the rigid function symbols followed by `_1` are introduced by the loop invariant rule and refer to corresponding field resp. array element value after the last loop iteration. Up to further notice all line numbers in the following paragraphs refer to Fig. 7.6

The base case establishes both properties for the start node itself (only node with path length 0). This branch can be closed nearly immediately by using the implication of lines 35-40 of the loop invariant. The antecedent of the implication can be proven true as the `current` field refers to `startNode` when leaving the loop.

For the step case, the things are more difficult. The principal idea is as follows: by induction hypothesis one knows that all nodes with a distance of iv have been visited and their children arrays are as before. In order to derive the same for a node m in distance $iv + 1$ one proceeds as follows: If m is also reachable within iv steps nothing has to be done and the induction hypothesis

KeY

```
\forall HeapObject start;\forall HeapObject ho;
  ((start != null & visited_1(start)=TRUE & ho != null &
  {\for (int i; HeapObject h)
      h.children[i] := get_1(h.children, i)}
    reach[\for (HeapObject x; int i) x.children[i];
          \for HeapObject x; x.children.length ](start, ho, iv))
  -> (visited_1(ho) = TRUE &
      \forall int i;(i>=0 & i<ho.children.length ->
                     get_1(ho.children,i) = childrenPre(ho,i))))
```

KeY

Figure 7.8: Use Case: Induction hypothesis

applies directly. Otherwise, there is a node p in distance iv from which m is reachable. The induction hypothesis allows to derive that iv is marked as visited and its children array is in its original state. Node p can then be used to instantiate the quantifier of line 21. The value of node p's `nextChild` field is equal the length of p's children array. Thus one has to use lines 26-32, if node m is referenced only by the last component of p's children entry, otherwise lines 22-24 have to be used. Now one knows that m is visited as well, the step case can then be closed by instantiating the quantifier in line 21 once again, but this time with m.

The use case can be closed by a direct instantiation and application of the induction hypothesis.

7.5 Results and Comparison

There is a variety of literature available about verification of the Schorr-Waite algorithm. We will briefly describe a (representative) selection of them.

Broy and Pepper The Schorr-Waite algorithm has been treated by [BP82]. In this paper, the authors start with the construction of an algebraical data type modelling a binary graph. They continue with the definition of the reflexive and transitive closure relation R^* of the graph. Then a function B is developed, *proven* to compute the set of all reachable graph nodes from a distinguished node x, i.e., $R^*(x)$. The function B turns out to realise the well-known depth-first traversal algorithm for (binary) graphs.

An extended graph structure is build upon the binary graph data type. In addition to the binary graph it provides two distinguished nodes (representing the current and previous node). Also two additional basic functions ex and rot are defined, which exchange the current and previous node resp. perform a rotation operation (forward step). By composition of these elementary graph operations a function is constructed that computes and return a tuple consisting of a set of nodes and an extended graph structure. It is proven that the returned

node set is the same as computed by the former function B and that the returned extended graph structure is the same on which the function has operated.

Afterwards the functional algorithm is refined to a procedural version.

Mehta and Nipkow The authors of [MN03] verify the correctness of a Schorr-Waite implementation (for binary graphs) using higher order logics. The program is written in a simple imperative programming language designed by the authors themselves. The operational semantics of the programming language has been modelled in Isabelle/HOL and a Hoare style calculus has been derived from the semantics.

The main difference to our approach is the explicit modelling of heaps and the distinction between addresses and references. On top of these definitions a reachability relation (and some auxiliary relations) is defined as above.

The program is then specified using Hoare logic by annotating the program with assertions and a loop invariant making use of the former defined relations. From these annotations, verification conditions are generated, which have to be proven by Isabelle/HOL.

Abrial The approach described in [Abr03] uses the B language and methodology to construct a correct implementation of the Schorr-Waite algorithm. Therefore the author starts with a high-level mathematical abstraction in B of a graph marking algorithm and then successively refines the abstraction towards an implementation of an (improved) version of Schorr-Waite. Each refinement step is accompanied by several proof-obligations that need to be proven to ensure the correctness of the refinement step.

H. Yang In [Yan00] the authors use a relative new kind of logic called Separation Logics, which is a variant of bunched implication logics. For verification they use a Hoare like calculus. The advantage of this logic is the possibility to express that two heaps are distinct and in particular the existence/possibility of a frame introduction rule. In short the frame introduction rule allows to embed a property shown for a local memory area in a global context with other memory cells.

The frame rule allows to show that if $\{P\}C\{Q\}$ is valid for a local piece of code C then one can embed this knowledge in a broader context $\{P * H\}C\{Q * H\}$ as long as the part of the heap H talks about is not altered by C (separate heaps). Without this frame rule one would have to consider H when proving $\{P\}C\{Q\}$, which makes correctness proves very tedious, in particular when the property shall be used in different *separate* contexts H_i.

Hubert and Marché In [HM05] the authors follow an approach very similar to the one presented in this chapter. They used a weakest precondition calculus for C implemented in the CADUCEUS tool to verify a C implementation of Schorr-Waite working on a bigraph. In the same manner as described here, they specified the loop invariant with help of an inductively defined reachable predicate using a higher order logic.

Part IV

Conclusions

8 Summary and Future Work

The previous chapters concentrated on specification and verification of programs implementing or using linked data structures like lists, trees or arbitrary graphs. The specification of graph related properties has been mainly build up on the notion of reachability.

In order to provide a notion suitable for specification and verification, several predicates have been introduced to specify those properties. These predicates had to be defined non-rigid as their truth value depends not only on the value of their arguments, but also on the current heap structure. It showed that classic non-rigid predicates lead to significant problems during the verification as the update simplification rules could not access knowledge on the locations on which a non-rigid predicate depended and had to assume safely that the truth value depended on all locations. The presented solution introduced a new class of non-rigid predicates encoding their dependencies into the symbol's syntax. These symbols proved not only useful for specification of graph data structure, but allowed also to improve the use of query symbols in the logic.

Besides the above mentioned specification predicates, a second way to specify linked data structure had been presented. Therefore abstract data types have been related to linked reference structures. As an elaborated example the presented technique had been applied to model Java-Strings.

Verification in presence of linked data structures includes often the verification of recursive defined methods. As Java Card DL has no possibility to quantify about the method invocation stack like other approached this had not been possible up-to-now. The presented rule enables their verification.

The specification and verification of Schorr-Waite in the arbitrary graph version demonstrated the practical use of some of the above introduced concepts.

As future work remains to include decision procedures for graph structure within the Java Card DL calculus. A first step towards this will be to implement existing rewrite systems for selected linked data structure in the rule and strategy framework. Further it is possible and intended to extend the notion of location dependent symbols, so that the location descriptors can make use of the symbols argument. The way to go is to allow free variables in the location descriptors which can be bound to the symbols arguments.

A Specification Predicates for Linked Datastructures

A.1 List Specification Predicates

```
\predicates {
    end(SimpleList);
   \nonRigid
      reach[\for SimpleList z;(z.next@(SimpleList))]
                (SimpleList, SimpleList, int);

   \nonRigid
      disjointSimpleList[\for SimpleList z;
                (z.next@(SimpleList))](SimpleList,SimpleList);

   \nonRigid
      unsharedNodeSimpleList[\for SimpleList z;
                (z.next@(SimpleList))] (SimpleList);

   \nonRigid
      isolatedSimpleList[\for SimpleList z;
                (z.next@(SimpleList))] (SimpleList,SimpleList);

   \nonRigid
      isList[\for SimpleList z;
                (z.next@(SimpleList))](SimpleList);
}
```

```
// == Reachable Axioms and Auxiliary Rules ==
// In the printed reachable definition the endmarker has
// been already replaced by null resp. the query 'x = null'
reachableDefinition {
    \schemaVar \term SimpleList t1, t2;
    \schemaVar \term int n;
    \schemaVar \variables SimpleList z;

    \find(reach[\for (z) z.next@(SimpleList)](t1, t2, n))
    \replacewith(t1 = t2 & n = 0 |
        (t1 != null & n > 0 &
         reach[\for (z) z.next@(SimpleList)]
                   (t1.next@(SimpleList), t2, n-1)))
 };

reachableDefinitionBase {
    \schemaVar \term SimpleList t1, t2;
    \schemaVar \variables SimpleList z;

    \find(reach[\for (z) z.next@(SimpleList)](t1, t2, 0))
    \replacewith(t1 = t2)
 };

reachableDefinitionFalse {
    \schemaVar \term SimpleList t1, t2;
    \schemaVar \term int n;
    \schemaVar \variables SimpleList z;

    \assumes (n <= -1 ==>)
    \find(reach[\for (z) z.next@(SimpleList)](t1, t2, n))
    \sameUpdateLevel
    \replacewith(false)
 };

reachableDefinitionStep {
    \schemaVar \term SimpleList t1, t2;
    \schemaVar \term int n;
    \schemaVar \variables SimpleList z;

    \assumes (n>=1 ==> t1 = null)
    \find(reach[\for (z) z.next@(SimpleList)](t1, t2, n))
    \sameUpdateLevel
    \replacewith(reach[\for (z) z.next@(SimpleList)]
                          (t1.next@(SimpleList), t2, n-1))
 };
```

```
reachableDefinitionStepAlt {
    \schemaVar \term SimpleList t1, t2;
    \schemaVar \term int n;
    \schemaVar \variables SimpleList z;

    \assumes (n>=1 ==> t2 = null)
    \find(reach[\for (z) z.next@(SimpleList)](t1, t2, n))
    \sameUpdateLevel
    \replacewith(reach[\for (z) z.next@(SimpleList)]
                         (t1, t2.next@(SimpleList), n+1))
 };
// The interactive rules are the same as above using 'add'
// instead of 'assumes', otherwise proofs cannot be saved.
reachableDefinitionFalseInteractive {
    \schemaVar \term SimpleList t1, t2;
    \schemaVar \term int n;
    \schemaVar \variables SimpleList z;

    \find(reach[\for (z) z.next@(SimpleList)](t1, t2, n))
    \sameUpdateLevel
    \replacewith(false);
    \add (==> n < 0)
 };

reachableDefinitionStepInteractive {
    \schemaVar \term SimpleList t1, t2;
    \schemaVar \term int n;
    \schemaVar \variables SimpleList z;

    \find(reach[\for (z) z.next@(SimpleList)](t1, t2, n))
    \sameUpdateLevel
    \replacewith(reach[\for (z) z.next@(SimpleList)]
                         (t1.next@(SimpleList), t2, n-1));
    \add(==>n>0 & t1 != null)
 };

reachableDefinitionStepAltInteractive {
    \schemaVar \term SimpleList t1, t2;
    \schemaVar \term int n;
    \schemaVar \variables SimpleList z;

    \find(reach[\for (z) z.next@(SimpleList)](t1, t2, n))
    \sameUpdateLevel
    \replacewith(reach[\for (z) z.next@(SimpleList)]
                         (t1, t2.next@(SimpleList), n+1));
    \add (==>n>0 & t2 != null) };
```

```
//\includeFile "listReachablePredicates.key";

// Rules for finite lists

// List specific
pathInListUnique {
     \schemaVar \term SimpleList t1, t2, t3;
     \schemaVar \term int n;
     \schemaVar \variables SimpleList z;

     \assumes(reach[\for (z) z.next@(SimpleList)]
                         (t3, t2, n) ==>)
     \find(reach[\for (z) z.next@(SimpleList)]
                         (t1, t2, n) ==>)
     \add(t1 = t3 ==>);
     \add(==> isList[\for z;(z.next@(SimpleList))](t1) &
              isList[\for z;(z.next@(SimpleList))](t3))
};

pathLengthInFiniteListUnique {
     \schemaVar \term SimpleList t1, t2, t3;
     \schemaVar \term int n1, n2;
     \schemaVar \variables SimpleList z;

     \assumes(reach[\for (z) z.next@(SimpleList)]
                         (t1, t2, n1) ==>)
     \find(reach[\for (z) z.next@(SimpleList)]
                         (t1, t2, n2) ==>)
     \add(n1 = n2 ==>);
     \add(==> isList[\for z;(z.next@(SimpleList))](t1) &
             isList[\for z;(z.next@(SimpleList))](t2) )
};
```

```
// == Axioms for Spec. Predicates ==

endMarkerIsNull {
   \schemaVar \term SimpleList list;

   \find(end(list))
   \replacewith(list = null)
   \heuristics (simplify)
};


disjointAxiomSimpleList {
    \schemaVar \term SimpleList x,y;
    \schemaVar \variables int n,m;
    \schemaVar \variables SimpleList sl;
    \schemaVar \variables SimpleList z;

    \find(disjointSimpleList[\for z;(z.next@(SimpleList))](x,y))
    \varcond(\notFreeIn(n,x), \notFreeIn(m,y),
             \notFreeIn(sl,x,y))
    \replacewith(!end(x) & !end(y) & \forall sl;(
        (\exists n;reach[\for z;
               (z.next@(SimpleList))](x,sl,n) &
         \exists m;reach[\for z;
               (z.next@(SimpleList))](y,sl,m)) -> end(sl)))
};


unsharedNode {
   \schemaVar \term SimpleList x;
   \schemaVar \variables SimpleList sl;
   \schemaVar \variables int n;
   \schemaVar \variables SimpleList z;

   \find(unsharedNodeSimpleList[\for z;
                                (z.next@(SimpleList))](x))
   \varcond(\notFreeIn(sl,x), \notFreeIn(n,x))
   \replacewith(\forall sl;(
     \forall n;!reach[\for z;(z.next@(SimpleList))](sl,x,n)))
};
```

```
listDefinition {
    \schemaVar \term SimpleList x;
    \schemaVar \variables int size;
    \schemaVar \variables SimpleList z;

    \find (isList[\for z;(z.next@(SimpleList))](x))
    \varcond(\notFreeIn(size,x))
    // \replacewith(\exists size;\exists z;(reach[\for
    //        z;(z.next@(SimpleList))](x,z,size) & end(z)))
    \replacewith(\exists size;(reach[\for z;
                   (z.next@(SimpleList))](x,null,size)))
};

// derived rule
listDefinitionTransitive {
    \schemaVar \term SimpleList x, y;
    \schemaVar \term int size;
    \schemaVar \variables SimpleList z;

    \find (reach[\for z;(z.next@(SimpleList))](x, y, size)==>)
    \add(isList[\for z;(z.next@(SimpleList))](y)==>);
    \add(==>isList[\for z;(z.next@(SimpleList))](x))
};

// list is acyclic
listIsAcyclic {
    \schemaVar \term SimpleList x, y;
    \schemaVar \term int size;
    \schemaVar \variables SimpleList z;

    \find (reach[\for z;(z.next@(SimpleList))]
         (x.next@(SimpleList), x, size))
    \sameUpdateLevel
    \add(==>isList[\for z;(z.next@(SimpleList))](x));
    \replacewith(x = null)
};
```

```
// improve isolated(x) <->
//     all y,z (reach(x,z) & reach(y,z) -> reach(x,y))
isolatedSimpleListDefinition {
    \schemaVar \term SimpleList x,y;
    \schemaVar \variables int n,m;
    \schemaVar \variables SimpleList sl;
    \schemaVar \variables SimpleList z;

   \find(isolatedSimpleList[\for z;(z.next@(SimpleList))](x, y))
   \replacewith(
      disjointSimpleList[\for z;(z.next@(SimpleList))](x, y) &
      unsharedNodeSimpleList[\for z;(z.next@(SimpleList))](x) &
      unsharedNodeSimpleList[\for z;(z.next@(SimpleList))](y) )
};
```

B Schorr-Waite Sources

B.1 Implementation

```
public class HeapObject {
    // as starter we only consider graphs with an out-degree
    // of at most two
    private HeapObject[] children;

    // used by the graph marking algorithm to
    // keep track of the next child to visit and
    // if this heap object has been already visited
    private int nextChild;
    private boolean visited;

    /** creates a new heap object */
    public HeapObject() {
        nextChild = 0;
        visited = false;
        children = new HeapObject[0];
    }

    /** creates a new heap object */
    public HeapObject(HeapObject[] children) {
        nextChild = 0;
        visited = false;
        this.children = children;
    }

    /**
     * sets <code>obj</code> as <code>pos</code>-th child
     */
    public void setChild(int pos, HeapObject obj) {
        if (pos < 0 || pos >= children.length) {
            throw new IndexOutOfBoundsException();
        }
        children[pos] = obj;
    }
```

```
    /**
     * sets the 'visited' marker
     */
    public void setMark(boolean mark) {
        this.visited = mark;
    }

    /**
     * sets the tracker of the next child to be visited
     */
    public void incIndex() {
        this.nextChild++;
    }

    /**
     * true if the mark is set
     */
    public boolean isMarked() {
        return visited;
    }

    /**
     * true if there is a next child to be visited
     */
    public boolean hasNext() {
        return nextChild<children.length;
    }

    /**
     * returns the index of the next child to visit
     */
    public int getIndex() {
        return nextChild;
    }

    /** returns the <code>pos</code>-th child */
    public HeapObject getChild(int pos) {
        return children[pos];
    }

    /** returns the numbe of children at this node */
    public int getChildCount() {
        return children.length;
    }
}
```

```
/**
 * This class implements the graph marking algorithm
 * known as Schorr-Waite algorithm.
 */
public class SchorrWaite {

    private HeapObject previous;
    private HeapObject current;

    public void mark(HeapObject start) {
        previous = null;
        current = start;
        current.setMark(true);
        while (start != current || start.hasNext()) {
            final HeapObject tmpChild;
            if (current.hasNext()) {
                final int nextChild = current.getIndex();
                tmpChild = current.getChild(nextChild);
                if (!tmpChild.isMarked()) {// forward scan
                    current.setChild(nextChild, previous);
                    previous = current;
                    current.incIndex();
                    current = tmpChild;
                    current.setMark(true);
                } else {
                    current.incIndex();
                }
            } else {
                // backward
                final int ref2restore = previous.getIndex() - 1;
                tmpChild = previous.getChild(ref2restore);
                previous.setChild(ref2restore, current);
                current = previous;
                previous = tmpChild;
            }
        }
    }
}
```

B.2 Schorr-Waite Proofobligation

```
\javaSource "../src/";

\sorts {
   \generic G;
}

\predicates{
  \nonRigid reach
           [\for (HeapObject x; int i)
                  x.children@(HeapObject)[i];
            \for (HeapObject x)
                  x.children@(HeapObject).length]
                              (Heapobject, HeapObject, int);
  \nonRigid onPath
           [\for (HeapObject x; int i)
                  x.children@(HeapObject)[i];
            \for (HeapObject x)
                  x.children@(HeapObject).length;
            \for (HeapObject x; int i)
                  x.nextChild@(HeapObject)]
                               (HeapObject, HeapObject, int);
}

\functions{
  HeapObject childrenPre(HeapObject, int);
}

\schemaVariables {
  \program Variable #a;
  \term HeapObject t, t1, t2, t3;
  \term int n, n1, n2;
  \term any then, else;
  \variables int k, iv, i, dist;
  \variables HeapObject u, hov, hov2, hov3;
}

\programVariables{
  SchorrWaite sw;
  HeapObject startNode;
}

\rules {

  ifExUniqueSolution {
     \find(\ifEx u; (u.#a = t.#a) \then (then) \else (else) )
     \varcond(\notFreeIn(u, t))
     \replacewith({\subst u; t}then )
     \heuristics (concrete)
  };
```

```
reachableDefinition {
  \find(reach[\for (hov; iv)
                hov.children@(HeapObject)[iv]@(HeapObject[]);
              \for (hov2)
                hov2.children@(HeapObject).length] (t1, t2, n))
  \varcond(\notFreeIn(k, t1, t2, n))
  \replacewith(t1 = t2 & n = 0 |
      (t1 != null & n > 0 & \exists k;(k>=0 &
            k<#lengthReference(t1.children@(HeapObject)) &
            reach[\for (hov; iv) hov.children@(HeapObject)[iv];
                 \for (hov2) hov2.children@(HeapObject).length]
                      (t1.children@(HeapObject)[k], t2, n-1))))
};
reachableDefinitionBase {
  \find(reach[\for (hov; iv)
                hov.children@(HeapObject)[iv]@(HeapObject[]);
              \for (hov2)
                hov2.children@(HeapObject).length](t1, t2, 0))
  \varcond(\notFreeIn(k, t1, t2))
  \replacewith(t1 = t2)
  \heuristics(simplify)
};
// on path
onPathDefinition {
 \find(onPath[\for (hov; iv)
                 hov.children@(HeapObject)[iv]@(HeapObject[]);
               \for (hov2) hov2.children@(HeapObject).length;
               \for (hov3) hov3.nextChild@(HeapObject)](t1, t2, n))
 \replacewith(n >= 0 & ((t1 = t2 & n = 0) |
    (t1 != null & t1.nextChild@(HeapObject) > 0 &
     t1.nextChild@(HeapObject) <=
          #lengthReference(t1.children@(HeapObject)) &
     onPath[\for (hov;iv) hov.children@(HeapObject)[iv]@(HeapObject[]);
            \for (hov2) hov2.children@(HeapObject).length;
            \for (hov3) hov3.nextChild@(HeapObject)]
       (t1.children@(HeapObject)[t1.nextChild@(HeapObject)-1],
        t2,n-1))))
};
onPathBase {
 \find(onPath[\for (hov; iv) hov.children@(HeapObject)[iv]@(HeapObject[]);
              \for (hov2) hov2.children@(HeapObject).length;
              \for (hov3) hov3.nextChild@(HeapObject)](t1, t2, 0))
 \replacewith(t1 = t2)
 \heuristics(simplify)
};
onPathNull {
 \find(onPath[\for(hov;iv)hov.children@(HeapObject)[iv]@(HeapObject[]);
              \for (hov2) hov2.children@(HeapObject).length;
              \for (hov3) hov3.nextChild@(HeapObject)](null, t2, n))
 \replacewith(n = 0 & t2 = null)
 \heuristics(simplify)
};
```

```
onPathEffectlessUpdate {
  \schemaVar \term int locIdx;
  \schemaVar \term HeapObject loc, val;
  \find(onPath[\for (hov; iv)
               hov.children@(HeapObject)[iv]@(HeapObject[]);
             \for (hov2) hov2.children@(HeapObject).length;
             \for (hov3) hov3.nextChild@(HeapObject)](t1, t2, n))
  \sameUpdateLevel
  \varcond(\notFreeIn(hov,loc),\notFreeIn(iv,n,t1,loc))
  \replacewith (
     {loc.children@(HeapObject)[locIdx]@(HeapObject[]):=val}
        onPath[\for (hov; iv)
                 hov.children@(HeapObject)[iv]@(HeapObject[]);
               \for (hov2) hov2.children@(HeapObject).length;
               \for (hov3) hov3.nextChild@(HeapObject)](t1, t2, n));
  \add(==> \forall iv;(iv>=0 & iv<=n ->
     !onPath[\for (hov; iv)
               hov.children@(HeapObject)[iv]@(HeapObject[]);
             \for (hov2) hov2.children@(HeapObject).length;
             \for (hov3) hov3.nextChild@(HeapObject)](t1, loc, n)));
  \add(==> \forall hov;(
     hov.children@(HeapObject)=loc.children@(HeapObject) -> hov=loc))
};

onPathEffectlessUpdate2 {
  \schemaVar \term int locIdx, val;
  \schemaVar \term HeapObject loc;

  \find(onPath[\for (hov; iv)
                 hov.children@(HeapObject)[iv]@(HeapObject[]);
               \for (hov2) hov2.children@(HeapObject).length;
               \for (hov3) hov3.nextChild@(HeapObject)](t1, t2, n))
  \sameUpdateLevel
  \varcond(\notFreeIn(iv,n,t1,loc))
  \replacewith (
     {loc.nextChild@(HeapObject):=val}
        onPath[\for (hov; iv)
                 hov.children@(HeapObject)[iv]@(HeapObject[]);
               \for (hov2) hov2.children@(HeapObject).length;
               \for (hov3) hov3.nextChild@(HeapObject)](t1, t2, n)
   )
   \add(\forall iv;(iv>=0 & iv<=n ->
       !onPath[\for (hov; iv)
                 hov.children@(HeapObject)[iv]@(HeapObject[]);
               \for (hov2) hov2.children@(HeapObject).length;
               \for (hov3) hov3.nextChild@(HeapObject)](t1,loc,n)) ==>);
   \add(==> \forall iv;(iv>=0 & iv<=n ->
       !onPath[\for (hov; iv)
                 hov.children@(HeapObject)[iv]@(HeapObject[]);
               \for (hov2) hov2.children@(HeapObject).length;
               \for (hov3) hov3.nextChild@(HeapObject)](t1,loc,n)))
 };
```

```
onPathNoCycle2 {
    \find(onPath[\for (hov; iv)
                    hov.children@(HeapObject)[iv]@(HeapObject[]);
                  \for (hov2) hov2.children@(HeapObject).length;
                  \for (hov3) hov3.nextChild@(HeapObject)]
        (t1, t1, n2) ==>)
     \varcond(\notFreeIn(i, t1))
     \add((t1 = null & n2 = 0) | n2 = 0 ==> );
     \add(==> \exists i;
        onPath[\for (hov; iv)
                  hov.children@(HeapObject)[iv]@(HeapObject[]);
                \for (hov2) hov2.children@(HeapObject).length;
                \for (hov3) hov3.nextChild@(HeapObject)](t1, null, i))
 };

 onPathTransitive2 {
   \find(onPath[\for (hov; iv)
                   hov.children@(HeapObject)[iv]@(HeapObject[]);
                 \for (hov2) hov2.children@(HeapObject).length;
                 \for (hov3) hov3.nextChild@(HeapObject)]
       (t1, t2, n2) ==>)
  \varcond(\notFreeIn(i, t1, t2, t3, n2))
  \add(\forall i;
     (onPath[\for (hov; iv)
                hov.children@(HeapObject)[iv]@(HeapObject[]);
              \for (hov2) hov2.children@(HeapObject).length;
              \for (hov3) hov3.nextChild@(HeapObject)](t2, t3, i)
    ->
      onPath[\for (hov; iv)
              hov.children@(HeapObject)[iv]@(HeapObject[]);
              \for (hov2) hov2.children@(HeapObject).length;
              \for (hov3) hov3.nextChild@(HeapObject)](t1,t3, n2 + i))
  ==>)
 };
}

\problem {
    inReachableState
  & !sw = null & !startNode = null
  & sw.<created> = TRUE & startNode.<created> = TRUE
  & \forall HeapObject ho;
      (!ho = null
         -> (ho.nextChild >= 0 & ho.nextChild <= ho.children.length))
  & \forall HeapObject ho;
           (!ho = null -> ho.children.length >= 0)
  & \forall HeapObject ho;
      (!ho = null -> !ho.children = null)
  & \forall HeapObject h1;
      \forall HeapObject h2;
      (!h1 = h2
        -> !h1.children = h2.children)
  & \forall HeapObject ho;
      \forall int i;
```

```
      (!ho = null & 0<=i & i< ho.children.length
                                          -> !ho.children[i] = null)
   & \forall HeapObject ho;
      \forall int i;
         ho.children[i] = childrenPre(ho, i)
   & \forall HeapObject ho;
     (!ho = null
       ->  ho.visited = FALSE
         & ho.nextChild = 0)
-> \[{
      sw.mark(startNode);
     }\] \forall HeapObject x; \forall int n;
      (!x = null &
          reach[\for (HeapObject x; int i) x.children@(HeapObject)[i];
                   \for HeapObject x; x.children@(HeapObject).length]
              (startNode, x, n)
         -> (x.visited = TRUE &
               \forall int i;(i>=0 & i<x.children.length ->
                                 x.children[i] = childrenPre(x,i))))
}
```

B.3 Soundness Proofobligations for derived Taclets

B.3.1 Taclet: EffectlessUpdate

```
\javaSource "../src/";

\sorts {
   \generic G;
}

\predicates{
  \nonRigid onPath[\for (HeapObject x; int i) x.children@(HeapObject)[i];
                \for (HeapObject x) x.children@(HeapObject).length;
                \for (HeapObject x; int i) x.nextChild@(HeapObject)]
          (HeapObject, HeapObject, int);
}

\schemaVariables {
  \term HeapObject t1, t2;
  \term int n;
  \variables int iv;
  \variables HeapObject hov, hov2, hov3;
}

\rules {
// on path

 onPathDefinition {
    \find ( onPath[\for (hov; iv)
                hov.children@(HeapObject)[iv]@(HeapObject[]);
                 \for (hov2) hov2.children@(HeapObject).length;
                 \for (hov3) hov3.nextChild@(HeapObject)](t1, t2, n) )
    \replacewith( n >= 0 &
         ( ( t1 = t2 & n = 0) |
              ( t1 != null & t1.nextChild@(HeapObject) > 0 &
                  t1.nextChild@(HeapObject) <=
                       #lengthReference(t1.children@(HeapObject)) &
                    onPath[\for (hov; iv)
                        hov.children@(HeapObject)[iv]@(HeapObject[]);
                           \for (hov2) hov2.children@(HeapObject).length;
                           \for (hov3) hov3.nextChild@(HeapObject)]
                                 (t1.children@(HeapObject)
                        [t1.nextChild@(HeapObject) - 1], t2, n-1) ) ) )
  };

}

\problem {
   \forall HeapObject t1;\forall HeapObject loc;\forall int n;(
         (  (\forall int i;(i>=0 & i<=n ->
                !onPath[\for (HeapObject x; int i) x.children@(HeapObject)[i];
                    \for (HeapObject x) x.children@(HeapObject).length;
                    \for (HeapObject x) x.nextChild@(HeapObject)]
               (t1, loc, i))) &
           \forall HeapObject t2;(t2.children = loc.children -> t2 = loc)) ->
   \forall HeapObject t2;\forall HeapObject val;\forall int locIdx;(
     (onPath[\for (HeapObject x; int i) x.children@(HeapObject)[i];
```

```
          \for (HeapObject x) x.children@(HeapObject).length;
          \for (HeapObject x) x.nextChild@(HeapObject)]
      (t1, t2, n)

   <-> {loc.children@(HeapObject)[locIdx]@(HeapObject[]):=val}
        onPath[\for (HeapObject x; int i) x.children@(HeapObject)[i];
          \for (HeapObject x) x.children@(HeapObject).length;
          \for (HeapObject x) x.nextChild@(HeapObject)]
        (t1, t2, n))))
}
```

B.3.2 Taclet: EffectlessUpdate2

```
\javaSource "../src/";

\sorts {
   \generic G;
}

\predicates{
  \nonRigid onPath[\for (HeapObject x; int i) x.children@(HeapObject)[i];
                   \for (HeapObject x) x.children@(HeapObject).length;
                   \for (HeapObject x; int i) x.nextChild@(HeapObject)]
           (HeapObject, HeapObject, int);
}

\schemaVariables {
  \term HeapObject t1, t2;
  \term int n;
  \variables int iv;
  \variables HeapObject hov, hov2, hov3;
}

\rules {
// on path

  onPathDefinition {
       \find ( onPath[\for (hov; iv)
                       hov.children@(HeapObject)[iv]@(HeapObject[]);
                      \for (hov2) hov2.children@(HeapObject).length;
                      \for (hov3) hov3.nextChild@(HeapObject)](t1, t2, n) )
       \replacewith( n >= 0 &
                ( ( t1 = t2 & n = 0) |
                     ( t1 != null & t1.nextChild@(HeapObject) > 0 &
                    t1.nextChild@(HeapObject) <=
                        #lengthReference(t1.children@(HeapObject)) &
                    onPath[\for (hov; iv)
                             hov.children@(HeapObject)[iv]@(HeapObject[]);
                           \for (hov2) hov2.children@(HeapObject).length;
                           \for (hov3) hov3.nextChild@(HeapObject)]
                                 (t1.children@(HeapObject)
                       [t1.nextChild@(HeapObject) - 1], t2, n-1) ) ) )
  };

}

\problem {
   \forall HeapObject t1;\forall HeapObject loc;\forall int n;(
          (\forall int i;(i>=0 & i<=n ->
              !onPath[
                 \for (HeapObject x; int i) x.children@(HeapObject)[i];
                 \for (HeapObject x) x.children@(HeapObject).length;
                 \for (HeapObject x) x.nextChild@(HeapObject)]
               (t1, loc, i))) ->
   \forall HeapObject t2;\forall int val;  (
     (onPath[\for (HeapObject x; int i) x.children@(HeapObject)[i];
          \for (HeapObject x) x.children@(HeapObject).length;
          \for (HeapObject x) x.nextChild@(HeapObject)]
```

```
      (t1, t2, n)
   <-> {loc.nextChild@(HeapObject):=val}
        onPath[\for (HeapObject x; int i) x.children@(HeapObject)[i];
          \for (HeapObject x) x.children@(HeapObject).length;
          \for (HeapObject x) x.nextChild@(HeapObject)]
        (t1, t2, n))))
}
```

B.3.3 Taclet: onPathBase

```
\javaSource "../src/";

\sorts {
   \generic G;
}

\predicates{
  \nonRigid onPath[\for (HeapObject x; int i) x.children@(HeapObject)[i];
                   \for (HeapObject x) x.children@(HeapObject).length;
                   \for (HeapObject x; int i) x.nextChild@(HeapObject)]
           (HeapObject, HeapObject, int);
}

\schemaVariables {
  \term HeapObject t1, t2, t3;
  \term int n;
  \variables int iv, i;
  \variables HeapObject hov, hov2, hov3;
}

\rules {

// on path

onPathDefinition {
 \find(onPath[\for (hov; iv)
                hov.children@(HeapObject)[iv]@(HeapObject[]);
                \for (hov2) hov2.children@(HeapObject).length;
                \for (hov3) hov3.nextChild@(HeapObject)](t1, t2, n) )
 \replacewith( n >= 0 &
    ((t1 = t2 & n = 0) |
      (t1 != null & t1.nextChild@(HeapObject) > 0 &
       t1.nextChild@(HeapObject) <=
          #lengthReference(t1.children@(HeapObject)) &
       onPath[\for (hov; iv) hov.children@(HeapObject)[iv]@(HeapObject[]);
              \for (hov2) hov2.children@(HeapObject).length;
              \for (hov3) hov3.nextChild@(HeapObject)](t1.children@(HeapObject)
                   [t1.nextChild@(HeapObject)-1], t2, n-1))))
};


/*
To be proven correct:
  onPathBase {
     \find(onPath[\for (hov; iv) hov.children@(HeapObject)[iv]@(HeapObject[]);
                  \for (hov2) hov2.children@(HeapObject).length;
                  \for (hov3) hov3.nextChild@(HeapObject)](t1, t2, 0) )
     \replacewith( t1 = t2 )
     \heuristics(simplify)
  };
*/
}
```

```
\problem {
   \forall HeapObject t1;\forall HeapObject t2;(
      onPath[\for (HeapObject x; int i) x.children@(HeapObject)[i];
           \for (HeapObject x) x.children@(HeapObject).length;
           \for (HeapObject x) x.nextChild@(HeapObject)]
        (t1, t2, 0) <-> t1 = t2)

}
```

B.3.4 Taclet: onPathNoCycle

```
\javaSource "../src/";

\sorts {
   \generic G;
}

\predicates{
  \nonRigid onPath[\for (HeapObject x; int i) x.children@(HeapObject)[i];
                  \for (HeapObject x) x.children@(HeapObject).length;
                  \for (HeapObject x; int i) x.nextChild@(HeapObject)]
           (HeapObject, HeapObject, int);
}

\schemaVariables {
  \term HeapObject t1, t2, t3;
  \term int n;
  \variables int iv, i;
  \variables HeapObject hov, hov2, hov3;
}

\rules {

// on path

onPathDefinition {
  \find ( onPath[\for (hov; iv) hov.children@(HeapObject)[iv]@(HeapObject[]);
                   \for (hov2) hov2.children@(HeapObject).length;
                   \for (hov3) hov3.nextChild@(HeapObject)](t1, t2, n) )
  \replacewith(n >= 0 &
      ((t1 = t2 & n = 0) | ( t1 != null & t1.nextChild@(HeapObject) > 0 &
        t1.nextChild@(HeapObject) <=
            #lengthReference(t1.children@(HeapObject)) &
        onPath[\for (hov; iv) hov.children@(HeapObject)[iv]@(HeapObject[]);
                \for (hov2) hov2.children@(HeapObject).length;
                \for (hov3) hov3.nextChild@(HeapObject)]
        (t1.children@(HeapObject)[t1.nextChild@(HeapObject)-1], t2, n-1))))
};


/*
To be proven correct:
onPathNoCycle2 {
  \find(onPath[\for (hov; iv) hov.children@(HeapObject)[iv]@(HeapObject[]);
                \for (hov2) hov2.children@(HeapObject).length;
                \for (hov3) hov3.nextChild@(HeapObject)](t1, t1, n) ==> )
  \varcond(\notFreeIn(i, t1))
  \add( (t1 = null & n = 0) | n = 0 ==> );
  \add( ==> \exists i;
       onPath[\for (hov; iv) hov.children@(HeapObject)[iv]@(HeapObject[]);
                \for (hov2) hov2.children@(HeapObject).length;
                \for (hov3) hov3.nextChild@(HeapObject)](t1, null, i))
  };
*/
```

```
onPathTransitive2 {
   \find(onPath[\for (hov; iv) hov.children@(HeapObject)[iv]@(HeapObject[]);
                \for (hov2) hov2.children@(HeapObject).length;
                \for (hov3) hov3.nextChild@(HeapObject)](t1, t2, n) ==> )
   \varcond(\notFreeIn(i, t1, t2, t3, n))
   \add(\forall i;(
       onPath[\for (hov; iv) hov.children@(HeapObject)[iv]@(HeapObject[]);
              \for (hov2) hov2.children@(HeapObject).length;
              \for (hov3) hov3.nextChild@(HeapObject)](t2, t3, i)
      ->
       onPath[\for (hov; iv) hov.children@(HeapObject)[iv]@(HeapObject[]);
               \for (hov2) hov2.children@(HeapObject).length;
               \for (hov3) hov3.nextChild@(HeapObject)](t1, t3, n+i)) ==>)
  };
}

\problem {
   \forall HeapObject t1;\forall int n;((
    \exists int i;(
       onPath[\for (HeapObject x; int i) x.children@(HeapObject)[i];
            \for (HeapObject x) x.children@(HeapObject).length;
            \for (HeapObject x) x.nextChild@(HeapObject)]
         (t1, null, i)) &
       !((t1 = null & n = 0) | n = 0)
    ) ->
     !(onPath[\for (HeapObject x; int i) x.children@(HeapObject)[i];
            \for (HeapObject x) x.children@(HeapObject).length;
            \for (HeapObject x) x.nextChild@(HeapObject)]
       (t1, t1, n)))
}
```

B.3.5 Taclet: onPathNull

```
\javaSource "../src/";

\sorts {
   \generic G;
}

\predicates{
  \nonRigid onPath[\for (HeapObject x; int i) x.children@(HeapObject)[i];
                \for (HeapObject x) x.children@(HeapObject).length;
                \for (HeapObject x; int i) x.nextChild@(HeapObject)]
          (HeapObject, HeapObject, int);
}

\schemaVariables {
  \term HeapObject t1, t2, t3;
  \term int n;
  \variables int iv, i;
  \variables HeapObject hov, hov2, hov3;
}

\rules {

// on path

onPathDefinition {
   \find(onPath[\for (hov; iv) hov.children@(HeapObject)[iv]@(HeapObject[]);
                 \for (hov2) hov2.children@(HeapObject).length;
                 \for (hov3) hov3.nextChild@(HeapObject)](t1, t2, n) )
   \replacewith( n >= 0 &
        ((t1 = t2 & n = 0) |
         (t1 != null & t1.nextChild@(HeapObject) > 0 &
          t1.nextChild@(HeapObject) <=
                #lengthReference(t1.children@(HeapObject)) &
          onPath[\for (hov; iv)
                          hov.children@(HeapObject)[iv]@(HeapObject[]);
                 \for (hov2) hov2.children@(HeapObject).length;
                 \for (hov3) hov3.nextChild@(HeapObject)]
         (t1.children@(HeapObject)[t1.nextChild@(HeapObject)-1],t2,n-1))))
  };


/*
To be proven correct:
  onPathNull {
     \find(onPath[\for (hov; iv) hov.children@(HeapObject)[iv]@(HeapObject[]);
                   \for (hov2) hov2.children@(HeapObject).length;
                   \for (hov3) hov3.nextChild@(HeapObject)](null, t2, n) )
     \replacewith( n = 0 & t2 = null )
     \heuristics(simplify)
  };
*/
}
```

```
\problem {
   \forall HeapObject t1;\forall int n;(
      onPath[\for (HeapObject x; int i) x.children@(HeapObject)[i];
           \for (HeapObject x) x.children@(HeapObject).length;
           \for (HeapObject x) x.nextChild@(HeapObject)]
        (null, t1, n) <-> (n=0 & t1 = null))

}
```

B.3.6 Taclet: onPathTransitive

```
\javaSource "../src/";

\sorts {
   \generic G;
}

\predicates{
  \nonRigid onPath[\for (HeapObject x; int i) x.children@(HeapObject)[i];
                  \for (HeapObject x) x.children@(HeapObject).length;
                  \for (HeapObject x; int i) x.nextChild@(HeapObject)]
           (HeapObject, HeapObject, int);
}

\schemaVariables {
  \term HeapObject t1, t2, t3;
  \term int n;
  \variables int iv, i;
  \variables HeapObject hov, hov2, hov3;
}

\rules {

onPathDefinition {
  \find ( onPath[\for (hov; iv) hov.children@(HeapObject)[iv]@(HeapObject[]);
                   \for (hov2) hov2.children@(HeapObject).length;
                   \for (hov3) hov3.nextChild@(HeapObject)](t1, t2, n) )
  \replacewith(n >= 0 &
      ((t1 = t2 & n = 0) | ( t1 != null & t1.nextChild@(HeapObject) > 0 &
        t1.nextChild@(HeapObject) <=
            #lengthReference(t1.children@(HeapObject)) &
        onPath[\for (hov; iv) hov.children@(HeapObject)[iv]@(HeapObject[]);
                \for (hov2) hov2.children@(HeapObject).length;
                \for (hov3) hov3.nextChild@(HeapObject)]
        (t1.children@(HeapObject)[t1.nextChild@(HeapObject)-1], t2, n-1))))
};

/*
 To be proven correct:
  onPathTransitive2 {
     \find(onPath[\for (hov; iv) hov.children@(HeapObject)[iv]@(HeapObject[]);
                    \for (hov2) hov2.children@(HeapObject).length;
                    \for (hov3) hov3.nextChild@(HeapObject)](t1, t2, n) ==> )
      \varcond(\notFreeIn(i, t1, t2, t3, n))
      \add(\forall i;
           (onPath[\for (hov; iv) hov.children@(HeapObject)[iv]@(HeapObject[]);
                  \for (hov2) hov2.children@(HeapObject).length;
                  \for (hov3) hov3.nextChild@(HeapObject)](t2, t3, i)
           ->
            onPath[\for (hov; iv) hov.children@(HeapObject)[iv]@(HeapObject[]);
                   \for (hov2) hov2.children@(HeapObject).length;
                   \for (hov3) hov3.nextChild@(HeapObject)](t1, t3, n + i))
        ==>)
  };
*/
}
```

```
\problem {

\forall HeapObject t1;\forall HeapObject t2;
  \forall int n;
    (     onPath[\for (HeapObject x; int i) x.children@(HeapObject)[i];
                 \for HeapObject x; x.children@(HeapObject).length;
                 \for HeapObject x; x.nextChild@(HeapObject)
                 ](t1, t2, n)
       -> \forall HeapObject t3;\forall int i;(
            onPath[\for (HeapObject x; int i) x.children@(HeapObject)[i];
                   \for HeapObject x; x.children@(HeapObject).length;
                   \for HeapObject x; x.nextChild@(HeapObject)
                  ](t2, t3, i) ->
            onPath[\for (HeapObject x; int i) x.children@(HeapObject)[i];
                   \for HeapObject x; x.children@(HeapObject).length;
                   \for HeapObject x; x.nextChild@(HeapObject)
                  ](t1, t3, n+i)
    ))
}
```

B.3.7 Taclet: reachableBase

```
\javaSource "../src/";

\sorts {
   \generic G;
}

\predicates{
  \nonRigid reach[\for (HeapObject x; int i) x.children@(HeapObject)[i];
                  \for (HeapObject x) x.children@(HeapObject).length]
            (HeapObject, HeapObject, int);
}

\schemaVariables {
  \term HeapObject t1, t2;
  \term int n;
  \term any then, else;
  \variables int k, iv;
  \variables HeapObject hov, hov2;
}

\rules {
 reachableDefinition {
      \find(reach[\for (hov; iv) hov.children@(HeapObject)[iv]@(HeapObject[]);
                  \for (hov2) hov2.children@(HeapObject).length](t1, t2, n))
      \varcond(\notFreeIn(k, t1, t2, n))
      \replacewith(t1 = t2 & n = 0 |
           (t1 != null & n > 0 &
            \exists k;( k>=0 & k<#lengthReference(t1.children@(HeapObject)) &
            reach[\for (hov; iv) hov.children@(HeapObject)[iv];
                  \for (hov2) hov2.children@(HeapObject).length]
               (t1.children@(HeapObject)[k], t2, n-1))) )
 };

/*
To be proven correct:
 reachableDefinitionBase {
      \find(reach[\for (hov; iv)
        hov.children@(HeapObject)[iv]@(HeapObject[]);
        \for (hov2) hov2.children@(HeapObject).length](t1, t2, 0))
      \varcond(\notFreeIn(k, t1, t2))
      \replacewith( t1 = t2 )
      \heuristics(simplify)
 };
*/
}

\problem {
   \forall HeapObject t1;\forall HeapObject t2;(
       reach[\for (HeapObject x; int i) x.children@(HeapObject)[i];
            \for (HeapObject x) x.children@(HeapObject).length]
         (t1, t2, 0) <-> t1 = t2)
}
```

Bibliography

[Abr03] J.R. Abrial. Event Based Sequential Program Development: Application to Constructing a Pointer Program. In *FME 2003: Formal Methods*, pages 51–74. Springer, September 2003.

[ARR03] Alessandro Armando, Silvio Ranise, and Michaël Rusinowitch. A rewriting approach to satisfiability procedures. *Inf. Comput.*, 183(2):140–164, 2003.

[Bec01] Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, volume 2041 of *LNCS*, pages 6–24. Springer-Verlag, 2001.

[BGH+04] Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Andreas Roth, Philipp Rümmer, and Steffen Schlager. Taclets: A new paradigm for constructing interactive theorem provers. *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales, Serie A: Matemáticas (RACSAM)*, 98(1), 2004. Special Issue on Symbolic Computation in Logic and Artificial Intelligence.

[BHS06] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer, 2006. To appear.

[BP82] Manfred Broy and Peter Pepper. Combining algebraic and algorithmic reasoning: An approach to the schorr-waite algorithm. *ACM Trans. Program. Lang. Syst.*, 4(3):362–381, 1982.

[BP06] Bernhard Beckert and André Platzer. Dynamic logic with non-rigid functions: A basis for object-oriented program verification. In U. Furbach and N. Shankar, editors, *Proceedings, International Joint Conference on Automated Reasoning, Seattle, USA*, LNCS 4130, pages 266–280. Springer, 2006.

[BRR04] Richard Bubel, Andreas Roth, and Philipp Rümmer. Ensuring correctness of lightweight tactics for java card dynamic logic. In *Preliminary Proceedings of Workshop on Logical Frameworks and Meta-Languages (LFM) at IJCAR 2004*, pages 84–105, 2004.

[BRS+00] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, volume 1783 of *LNCS*. Springer-Verlag, 2000.

[Bub01] Richard Bubel. Behandlung der Initialisierung von Klassen und Objekten in einer dynamischen Logik für Java Card. Studienarbeit, Fakultät für Informatik, Universität Karlsruhe, August 2001.

[Gie98] Martin Giese. Integriertes automatisches und interaktives Beweisen: Die Kalkülebene. Diploma Thesis, Fakultät für Informatik, Universität Karlsruhe, June 1998.

[Gie03] Martin Giese. Taclets and the KeY prover. In *User Interfaces for Theorem Provers Workshop at TPHOLS, Rome, Italy*, 2003.

[Gie05] M. Giese. A calculus for type predicates and type coercion. In B. Beckert, editor, *Automated Reasoning with Analytic Tableaux and Related Methods, Tableaux 2005*, volume 3702 of *LNAI*, pages 123–137. Springer, 2005.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[GJSB04] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification–Third Edition*. The Java Series. Addison-Wesley,, 2004.

[HÖ5] Reiner Hähnle. Many-valued logic, partiality, and abstraction in formal specification languages. *Logic Journal of the IPGL*, 13(4):415–433, july 2005.

[Har84] David Harel. Dynamic logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume II: Extensions of Classical Logic, chapter 10, pages 497–604. Reidel, Dordrecht, 1984.

[HM05] Thierry Hubert and Claude Marché. A case study of C source code verification: the Schorr-Waite algorithm. In Bernhard K. Aichernig and Bernhard Beckert, editors, *Proc. Third IEEE International Conference on Software Engineering and Formal Methods (SEFM), Koblenz, Germany*, pages 190–199. IEEE Computer Society, 2005.

[LBR00] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i, Iowa State University, Department of Computer Science, February 2000.
`ftp://ftp.cs.iastate.edu/pub/techreports/TR98-06/TR.ps.gz`.

[LPC+02] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, and Clyde Ruby. *JML Reference Manual*, August 2002.

[MN03] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. In F. Baader, editor, *Automated Deduction — CADE-19*, volume 2741 of *LNCS*, pages 121–135. Springer, 2003.

[Obj01] Object Modeling Group. *Unified Modelling Language Specification, Version 1.4*, September 2001.

[Ohe01] David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[Pau94] Lawrence C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *LNCS*. Springer-Verlag, 1994.

[Rot06] Andreas Roth. *Specification and Verification of Object-oriented Components*. PhD thesis, Fakultät für Informatik der Universität Karlsruhe, June 2006.

[Rüm06] Philipp Rümmer. Sequential, parallel, and quantified updates of first-order structures. In *Logic for Programming, Artificial Intelligence and Reasoning*, volume 4246 of *LNCS*, pages 422–436. Springer-Verlag, 2006.

[Sch07] Steffen Schlager. *TBA*. PhD thesis, Fakultät für Informatik der Universität Karlsruhe, February 2007.

[SRW99] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3–valued logic. In *Symposium on Principles of Programming Languages*, pages 105–118, 1999.

[SW67] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM*, 10(8):501–506, 1967.

[Tre05] Kerry Trentelman. Proving correctness of javacard dl taclets using bali. In *SEFM*, pages 160–169, 2005.

[vdBJ00] J. van den Berg and BPF Jacobs. *The LOOP Compiler for Java and JML*. Computing Science Institute Nijmegen, Faculty of Science, University of Nijmegen, 2000.

[Wid06] Florian Widmann. Crossverification of while loop semantics. Diplomarbeit, Universität Karlsruhe, Fakultät für Informatik, October 2006.

[Yan00] H. Yang. An example of local reasoning in bi pointer logic: the schorr-waite graph marking algorithm, 2000.