Deductive Verification of Safety-Critical Java Programs

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

der Fakultät für Informatik
der Universität Fridericana zu Karlsruhe (TH)

genehmigte

Dissertation

von

Christian Engel

aus Karlsruhe

Tag der mündlichen Prüfung:	13.11.2009
Erster Gutachter:	Prof. Dr. P. H. Schmitt, Universität Karlsruhe (TH)
Zweiter Gutachter:	Prof. assoc. Dr. B. Thomsen, Aalborg University

Acknowledgements

This thesis would not have been possible without the support and encouragement of many people to whom I would like to express my gratitude here.

In the first place, I would like to sincerely thank my supervisor, Prof. Dr. Peter H. Schmitt for giving me the opportunity to work in his research group and for his valuable support and guidance.

I am grateful to Prof. Dr. Bent Thomsen for acting as second reviewer of my thesis.

I also would like to show my gratitude to Prof. Dr. Ralf Reussner and Dr. Carsten Sinz for their interest in my work and agreeing to act as examiners for the thesis defense.

A special thank goes to my colleagues and former colleagues Daniel Bruns, David Farago, Florian Merz, Dr. Hendrik Post, Dr. Steffen Schlager, Mattias Ulbrich, Dr. Olga Tveretina, Benjamin Weiß and Dr. Frank Werner for providing a relaxed working atmosphere.

Moreover, I would like to thank all active and former members of the KeY project; in particular my thanks go to Dr. Richard Bubel, Prof. Dr. Reiner Hähnle, Dr. Andreas Roth and Dr. Philipp Rümmer.

I owe my deepest gratitude to my parents - Helga and Manfred Engel - for many years of support during my studies without which this thesis would not have been possible. Finally, I would like to deeply thank Ulrike for her love, encouragement and support.

Karlsruhe, November 2009
Christian Engel

Deduktive Verifikation sicherheitskritischer Java-Programme

(Deutsche Zusammenfassung)

Die vorliegende Arbeit entstand im Rahmen des KeY-Projekts, das sich mit der Integration formaler Methoden in den Softwareentwicklungsprozeß befaßt. Ein großer Teil der in dieser Arbeit vorgestellten Ansätze wurde implementiert und in das KeY-System integriert, einem aus dem KeY-Projekt hervorgegangenen Werkzeug zur deduktiven formalen Verifikation von Java Card-Programmen mittels *symbolischer Ausführung*. Hierbei werden die dem Programm zur Verfügung stehenden Speicherstellen, wie beispielsweise lokale Variablen, nicht wie bei der tatsächlichen Programmausführung mit konkreten Werten belegt, sondern mit symbolischen. Entsprechend wird anstelle eines konkreten Programmdurchlaufs die Menge aller für den symbolischen Startzustand[1] des Programms möglichen Programmdurchläufe und aller daraus resultierenden symbolischen Endzustände betrachtet. Die vom KeY-System zu beweisenden Korrektheitsaussagen werden in *dynamischer Logik* formalisiert, bei der es sich um eine Erweiterung der klassischen Prädikatenlogik erster Stufe handelt, die (ähnlich wie *Hoare Logik*) das Formulieren von Aussagen über Programmzustände und Übergänge zwischen diesen erlaubt.

Seit einigen Jahren gibt es Bestrebungen, Java echtzeitfähig zu machen. Als Ergebnis dieser Bestrebungen wurde im Jahr 2002 die *Real-Time Specification for Java* (RTSJ) veröffentlicht. Eine der wichtigsten der darin eingeführten Neuerungen ist ein Speichermodell, das die explizite Freigabe kompletter, zuvor definierter Speicherbereiche (sog. *Scoped Memory*) erlaubt. Dadurch wird der *Java Garbage Collector* überflüssig gemacht, der eines der Haupthindernisse für die Verwendung von Java in Echtzeitsystemen darstellt. Diese Arbeit beschäftigt sich nun mit verschiedenen Fragestellungen, die insbesondere bei der deduktiven Verifikationen von RTSJ-Programmen auftreten und diskutiert Lösungsansätze für diese.

Teil I führt in die zum Verständnis dieser Arbeit nötigen Grundlagen ein. Zunächst werden Syntax und Semantik der im KeY-System verwendeten Dynamischen Logik definiert, dann ausgewählte Aspekte (beispielsweise die Behandlung innerer und anonymer Klassen) eines auf dieser Logik basierenden und ebenfalls im KeY-System zum Einsatz kommenden Sequenzenkalküls besprochen. Des Weiteren enthält dieser Teil eine kurze Einführung in die Spezifikationssprache JML (*Java Modeling Language*), von der an diversen Stellen in dieser Arbeit Gebrauch gemacht wird, und einen Überblick über die für diese Arbeit relevanten Merkmale von RTSJ.

Der zweite Teil der Arbeit beschäftigt sich mit dem Speichermodell der RTSJ und sich daraus ergebenden Problemstellungen. Kapitel 5 definiert die im Folgenden betrachtete Untermenge der RTSJ und motiviert die vorgenommenen Einschränkungen. In Kapitel 6 wird eine Formalisierung des RTSJ-Speichermodells in dynamischer Logik vorgestellt und diese mit

[1] Dieser Startzustand kann Einschränkungen unterliegen (einer sog. Vorbedingung), die im KeY System als prädikatenlogische Formeln über den verwendeten symbolischen Werten ausgedrückt werden können.

einem alternativen Ansatz verglichen. Wie die in diesem Kapitel eingeführten Konzepte zur Spezifikation von RTSJ-Programmen genutzt werden können, wird beispielhaft in Kapitel 7 aufgezeigt. Dieses Kapitel geht auch auf Beweisverpflichtungen für RTSJ-Programme ein. Es gibt eine Reihe von auf der RTSJ basierenden Profilen für sicherheitskritische JAVA Anwendungen (SCJ[2].), die eine verbesserte Analysierbarkeit des Codes unter anderem durch Einschränkungen des Sprachumfangs herbeizuführen versuchen. Wie in Kapitel 8 erläutert, liegen Programme, die unter Berücksichtigung eines dieser gängigen SCJ Profile geschrieben wurden, in der (zuvor in Kapitel 5 definierten) von unserem Ansatz betrachteten Untermenge RTSJs. Andere SCJ-Konzepte sehen die implizite Bereitstellung neuer *Scoped Memory*-Bereiche bei Methodenaufrufen vor, deren Lebensdauer der Ausführungsdauer der aufgerufenen Methode entspricht[3]. Der vorgestellte Ansatz zur Behandlung des RTSJ-Speichermodells läßt sich auch für solche, die implizite Erzeugung von *Scoped Memory*-Bereichen vorsehenden Speichermodelle einsetzen. Kapitel 9 beleuchtet schließlich die Frage, wie man unter Ausnutzung des RTSJ-Speichermodells die Nichtinterferenz verschiedener nebenläufiger Programmfäden[4] (bewiesenermaßen) sicherstellen kann. Unter anderem werden dazu Beweisverpflichtungen in Dynamischer Logik für verschiedene Datenkapselungs- und Abhängigkeitseigenschaften vorgestellt (so z. B. für die *depends*-Klausel, die angibt, von den Werten welcher Speicherstellen das Verhalten einer Methode abhängt) und deren Korrektheit und (sofern zutreffend) Vollständigkeit bewiesen.

Teil III widmet sich der modularen vertragsbasierten Verifikation des Heap-Speicherverbrauchs von JAVA-Programmen, eine Thematik die von besonderer Relevanz hinsichtlich der RTSJ ist, da hier Speicherbereiche benutzt werden, die nicht durch den *Garbage Collector* bereinigt werden. Kapitel 11 faßt die bestehenden Konzepte zusammen, die die Spezifikationssprache JML zum Spezifizieren des Speicherverbrauchs von JAVA-Methoden bietet, und zeigt deren Defizite bezüglich der Modularisierbarkeit der Spezifikation (und entsprechend auch des Verifikationsprozesses) auf. Gleichzeitig werden Vorschläge gemacht, wie diese Defizite mit Hilfe einer Reihe neuer JML-Konstrukte zu beheben sind. Abschließend definiert dieses Kapitel, wie die neu eingeführten und die schon bestehenden JML-Konstrukte (zum Spezifizieren des Speicherverbrauchs) in der von KeY verwendeten dynamischen Logik repräsentiert werden. Daraufhin werden Beweisverpflichtungen (Kapitel 12) für die Korrektheit von Speicherverbrauchsspezifikationen und Erweiterungen des Kalküls (Kapitel 13) zur Behandlung der in Kapitel 11 eingeführten Symbole vorgestellt. Wesentliche Punkte sind hierbei die modulare Behandlung von Methodenaufrufen und die Verifikation des Speicherverbrauchs von Schleifen. Die Funktionsweise dieser Kalkülerweiterungen wird in Kapitel 14 anhand einiger Beispiele erläutert, unter anderem einer Methode aus der Echtzeit-JAVA-Bibliothek *Javolution*, die unter Verwendung der zuvor eingeführten JML-Konstrukte spezifiziert und mit Hilfe einer prototypischen Implementierung des vorgestellten Ansatzes im KeY-System verifiziert wurde. Das folgende Kapitel (15) faßt zusammen, welche Anpassungen des Kalküls zur Behandlung von Speichermodellen mit impliziter Erzeugung von *Scoped Memory*-Bereichen notwendig sind. Besonderes Augenmerk liegt hier auf dem von dem SCJ-Compiler PERC Pico verwendeten Speichermodell, das nicht nur die Erzeugung von mit Methodenausführungen assoziierten Speicherbereichen erlaubt sondern auch von solchen, die mit neu erzeugten Objekten assoziiert sind und deren Lebensdauer besitzen.

[2] Steht für *Safety Critical Java*

[3] Insofern ahmt dieser Ansatz in gewisser Weise Stackallokierung nach.

[4] engl.: *Threads*

Contents

List of Figures

List of Tables

1 Introduction

Computers have nowadays become a virtually omnipresent part of our daily lives. This goes along with an increasing use of computers, and consequently also software, in safety-critical systems such as medical equipment, aviation- and automotive control systems or nuclear power plants. The necessity for these systems to be correct is evident and often stringent certification standards (such as DO-178B [RTCA, 1992]) are established for ensuring this. In general, current certification standards for safety critical systems mainly rely on software reviews and testing both of which are time-consuming, expensive and usually cannot guarantee the correctness of software (i.e., compliance with its specification) with absolute certainty. Considering this, formal methods appear to be the obvious technology for decreasing certification costs as well as increasing the confidence in software correctness. Future certification standards (e.g., DO-178C) currently in the process of being defined are seemingly taking into account formal methods as an alternative to testing and reviewing software.

Most of the above mentioned examples for safety-critical systems must also meet real-time (RT) constraints (and are thus also called *real-time systems*). A RT system is characterized by responding to an event[1] in a defined time interval (in this context the end of this interval is referred to as a deadline). Missing a deadline renders the system response useless (*hard real-time*) or just less useful (*soft real-time*) compared to a response performed in time. An engine control unit can, for instance, be considered hard real-time: an ignition spark triggered too late is useless. In contrast a video decoder is a soft real-time application: if a deadline for decoding a video frame is missed it can still make sense to display the frame after expiration of its deadline. The "value" of doing so, however, decreases the more time has elapsed since the deadline has passed.

For analysing whether an RT system will always meet its deadlines it is essential that software employed in such a system is not only (functionally) correct, it must also provide deterministic performance. In this context, the relevant parameter is *worst-case* performance (as an RT system is required to *always* meet its deadline, even in the *worst case*) not *average* performance.

Though historically RT systems are mainly programmed in assembler, C or Ada, in recent years a trend to make Java suitable for real-time applications could be observed. Standard Java cannot be used for programming RT applications for several reasons: Its garbage collector, for instance, leads to indeterministic performance since it can interrupt a program at arbitrary occasions for unbounded periods of time. Therefore, the *Java Specification Request 1* (JSR 1) for a real-time version of Java was issued in 1998 resulting in the *Real-Time Specification for Java* (RTSJ, [Bollella and Gosling, 2000]). Among the improvements (with respect to RT compliance) introduced by the RTSJ is a region-based memory model featuring memory regions which can be explicitly freed and are not subject to garbage collection. In addition it provides a novel kind of threads only allowed to operate on objects allocated in

[1] Events do not need to be external stimuli but can also be time-triggered such as in the case of a periodic task.

explicitly freeable regions. Since these threads do not interfere with the garbage collector, they can interrupt it at arbitrary times.

This thesis elaborates on several aspects of deductive formal verification of RTSJ programs. One of the key contributions is an approach for treating the just mentioned RTSJ memory model. The formalism we use throughout this work for describing the presented approaches is Java DL a dynamic logic [Harel, 1984] for a sequential subset of Java and a sequent calculus for this logic. Most of the contributions of this thesis were implemented in the KeY system, a semi-automatic theorem prover for Java DL.

1.1 The KeY System

The KeY system [Beckert et al., 2007] is a semi-automatic theorem prover jointly developed by the University of Karlsruhe, Chalmers University of Technology in Gothenburg, and the University of Koblenz within the context of the KeY project (`http://www.key-project.org`), which aims for the integration of formal methods into the industrial software engineering process. Initially, KeY's target language was Java Card DL, a dynamic logic for Java Card [Chen, 2000] which is a Java dialect intended for applications running on smart cards. Meanwhile, however, KeY supports several other logics, such as a dynamic logic for *MISRA C* [Mürk et al., 2007], a logic for JCSP programs [Klebanov et al., 2005], a logic for a generic object oriented programming language called ODL [Platzer, 2004] and a Differential Dynamic Logic for Hybrid Systems [Platzer and Quesel, 2008]. Also the supported Java subset is now a real superset of Java Card including for instance multi-dimensional arrays and all primitive Java data types. There are also efforts to support Java 5 in KeY [Ulbrich, 2007]. Therefore, we will refer to Java Card DL as just Java DL in the remainder of this work as this terminology better fits its current characteristics.

KeY possesses frontends for the following specification languages: The *Java Modeling Language* JML [Engel, 2005] (which we make use of on several occasions in this work) and the *Object Constraint Language* (OCL) [Beckert et al., 2002], both of which are facilitating the *Design by Contract* [Meyer, 1992] paradigm. These frontends allow the generation of various kinds of Java DL proof obligations (such as a proof obligation for checking whether a method establishes the postcondition specified by its specification) for different correctness aspects of a given specification written in OCL or JML. The frontends also map JML and OCL method contracts to their Java DL counterparts which are required for modular program verification (for details, refer to Section 2.3.5).

There are also backends leveraging formal proofs and proof attempts performed by KeY for different purposes than formal reasoning: Proofs in the KeY system basically constitute a symbolic execution [King, 1976] tree (which is owed to employing a sequent calculus and symbolic execution) from which we can obtain execution path conditions and symbolic states. This information can serve as input for white-box test case generation [Engel and Hähnle, 2007, Beckert and Gladisch, 2007] or the visualisation [Baum, 2007, Rothe, 2008] of the symbolic execution tree including its path conditions and symbolic program states. The latter backend, called the *symbolic debugger*, can be used for debugging or just for adding to understanding the program's behavior.

As KeY aims at being integrated in a realistic software engineering process, it is not only available as a standalone version but also as plug-in to the Eclipse IDE [Holzner, 2004].

1.2 Outline

The thesis is structured in three parts:

Part I summarises the foundations required for the remainder of this work. The basic notational and semantic concepts of Java DL are introduced and solutions for handling inner classes and comprehensions are presented in Chapter 2. This is followed by overviews of the *Java Modeling Language* (JML, Chapter 3) and the *Real-Time Specification for Java* (RTSJ, Chapter 4).

Part II discusses the RTSJ's memory model and shows how it can be modeled in Java DL. Chapter 8 elaborates on how the presented approach can be adapted to treat other Java dialects (such as PERC Pico [Nilsen, 2006]) for safety critical applications based on implicit scope identities. Finally, in Chapter 9, several existing proof obligations for guaranteeing data encapsulation properties are reviewed and novel ones are proposed. We also consider how the RTSJ memory model can facilitate non-interference of threads.

Part III is focused on the contract-based verification of worst-case memory usage (WCMU) of Java programs (in particular RTSJ programs). The JML approach for specifying a method's WCMU is discussed in Chapter 11. We also point out deficiencies of the current JML approach and propose solutions overcoming these deficiencies. Chapter 13 provides a formal semantics for the afore introduced WCMU specification means by defining their handling in the Java DL calculus. The following Chapter illustrates the usage of this calculus by means of two example methods whose WCMU specification was verified using the described approach. Chapter 15 again discusses adaptations required by the PERC Pico memory model.

Chapter 17 concludes this thesis and points out directions of future work.

1.3 Contributions

The main contributions of this work can be summarised as follows:

- A calculus for sequential RTSJ programs is presented and its suitability also with regard to SCJ profiles is discussed.
- We elaborate on the ensurance of non-interference by leveraging the RTSJ memory model. In this context several proof obligations for *depends* and *captures* clauses are developed and proofs for their correctness and (if applicable) completeness are provided.
- An approach for modularly specifying and verifying *worst case memory usage* constraints is developed:

  A proposal for enhancing the JML specification means (for memory usage) is made, which facilitates the formulation of modular and precise memory consumption contracts. In addition a calculus for modularly verifying these contracts is provided.

Part I

Foundations

2 Java DL

The logic used in this work for reasoning about safety-critical Java programs is Java DL [Beckert et al., 2007] a dynamic first order logic for a sequential subset of Java. Dynamic first order logic as invented by David Harel [Harel, 1984] extends first order logic by two modal operators $[p]$ (box) and $\langle p \rangle$ (diamond) for every legal sequence p of statements in the regarded programming language. The intuitive meaning of a formula $[p]\psi$ is the following partial correctness statement[1]: *If p terminates then ψ holds in the post-state of p.* The diamond modality in addition requires termination thus $\langle p \rangle\psi$ means: *p terminates and ψ holds in the post-state of p* (i.e., total correctness with regard to the postcondition ψ). The commonly known Hoare triple $\{\phi\}p\{\psi\}$ is expressible in dynamic logic by the formula: $\phi \rightarrow [p]\psi$. In contrast to Hoare logic, however, dynamic logic is closed under logic and modal operators.

A particularity of Java DL not present in the basic form of dynamic logic ([Harel, 1984]) are updates. Updates are a means to encode state transitions and work basically as lazy substitutions. One important application of updates in Java DL is their usage as an intermediate language: When evaluating a formula $[p]\psi$, the program p is first "compiled" (which is done by symbolic execution) to an update $\{\mathcal{U}\}$ resulting in a formula $\{\mathcal{U}\}\psi$. The update $\{\mathcal{U}\}$ is then applied to ψ (meaning the state change it encodes is made explicit by applying appropriate substitutions to ψ). The advantage this approach carries is that case distinctions necessitated by the execution of p (e.g., when encountering **`if`**-statements) or by the computation of the state change (e.g., for treating aliasing) are encoded in the update and proof splits triggered by these case distinctions are postponed until the update is actually applied to a formula not having a modality as top-level operator.

The definition of Java DL we use here is mainly in line with [Beckert et al., 2007]. Thus we only discuss Java DL to the level of detail needed to achieve a sensible degree of self-containment of this work. Focus is put on those aspects of the calculus which are of special relevance for the remainder of this thesis (such as the handling of method calls), definitions and notations differing from previous works (anonymising updates, location descriptors) and issues not handled previously in Java DL (comprehensions, inner classes). For an exhaustive account of Java DL refer to [Beckert et al., 2007].

2.1 Syntax

2.1.1 Type Hierarchy

The first order fragment of Java DL is a typed first order logic. As its signature (see Section 2.1.2), its type hierarchy is not fixed but partly determined by the regarded program context. The type hierarchy of the program context is embedded (the type hierarchy of the program

[1] We assume here that the regarded programming language is deterministic otherwise, in the general case, the box operator $[p]$ can be considered a universal quantification over the reachable post-states of p whereas $\langle p \rangle$ is an existential quantification over these states.

context itself does not form a bounded lattice) in a bounded lattice forming the type system of Java DL.

Definition 2.1 (Type Hierarchy) *A* Java DL *type hierarchy is a bounded lattice* $(\mathcal{T}, \sqsubseteq)$ *where*

- $\mathcal{T}$ *is a finite set of types such that*
  - $\mathcal{T} = \mathcal{T}_a \cup \mathcal{T}_d$ *with* $\mathcal{T}_a$ *denoting the set of abstract types and* $\mathcal{T}_d$ *denoting the set of dynamic types.*
  - $\mathcal{T}_a \cap \mathcal{T}_d = \emptyset$
  - $\top \in \mathcal{T}_d$ *where* $\top$ *is also referred to as the type* any
  - $\bot \in \mathcal{T}_a$
- *The relation* $\sqsubseteq$ *is a partial order on* $\mathcal{T}$ *with* $\top$ *and* $\bot$ *being the top resp. bottom element of the lattice* $(\mathcal{T}, \sqsubseteq)$*, meaning:*

$$\bot \sqsubseteq A \sqsubseteq \top \textit{ f. a. } A \in \mathcal{T}$$

A dynamic type $D \in \mathcal{T}_d$ can have elements that are of exact type D (i.e., of type D and not a real subtype of it). In contrast, an abstract type $A \in \mathcal{T}_a$ can only have elements whose exact type is a real subtype of A. In the context of Java primitive and non-abstract class types are represented as dynamic types whereas interface and abstract class types are represented as abstract types.

2.1.2 Signature

The signature of Java DL also depends on the program context since, for instance, attributes are represented by function symbols. Since the interpretation of certain symbols (such as function symbols representing attributes) depends on the program state, we make a distinction between symbols which are state-dependent (a property we call *flexible* or *non-rigid* in the following) and those which are not state-dependent (*rigid*).

Definition 2.2 (Signature) *Let* $(\mathcal{T}, \sqsubseteq)$ *be a type* Java DL *hierarchy then a* Java DL *signature for* $(\mathcal{T}, \sqsubseteq)$ *is a tuple*

$$\Sigma := (VSym, FSym_r, FSym_{nr}, PSym_r, PSym_{nr}, \alpha)$$

where

- $VSym$ *is a set of (logic) variables*
- $FSym_r$ *is a set of rigid function symbols and* $FSym_{nr}$ *a set of non-rigid function symbols with* $FSym_r \cap FSym_{nr} = \emptyset$*. We define* $FSym := FSym_r \cup FSym_{nr}$*.*
- $PSym_r$ *is a set of rigid predicate symbols and* $PSym_{nr}$ *a set of non-rigid predicate symbols with* $PSym_r \cap PSym_{nr} = \emptyset$*. We define* $PSym := PSym_r \cup PSym_{nr}$*.*

- *α is a typing function fixing the signature of each symbol:*

$$\alpha : \{VSym, FSym, PSym\} \to \mathcal{T}^+$$

such that

- $\alpha(v) \in \mathcal{T}$ *for all* $v \in VSym$
- $\alpha(f) \in \mathcal{T}^{n_f} \times \mathcal{T}$ *for all* $f \in FSym$ *where* n_f *(with* $n_f \geq 0$*) is the arity of* f *and the last element in* $\alpha(f)$ *the result type of* f*.*
- $\alpha(P) \in \mathcal{T}^{n_P}$ *for all* $P \in PSym$ *where* n_P *(with* $n_P \geq 0$*) is the arity of* P*.*

For every Java DL signature several predefined and interpreted function and predicate symbols exist such as arithmetic operators, boolean constants (denoted *TRUE* and *FALSE* in the following) or function symbols stemming from the program context (e.g., attributes). A complete list of predefined Java DL symbols can be found in [Beckert et al., 2007].

2.1.3 Terms, Formulas and Updates

The syntax of the first order fragment of Java DL largely matches the standard syntax definitions of first order logic. This section therefore concentrates on Java DL specific constructs used in the remainder[2] of this work and typing issues.

Definition 2.3 (Terms) *Let* $(VSym, FSym_r, FSym_{nr}, PSym_r, PSym_{nr}, \alpha)$ *be a* Java DL *signature for a type hierarchy* $(\mathcal{T}, \sqsubseteq)$ *then* $Terms_T$ *(the terms of type* T*) with* $T \in \mathcal{T}$ *is the smallest set such that*

- $x \in Terms_T$ *for all* $x \in VSym$ *with* $\alpha(x) = T$
- $f(t_1, \ldots, t_n) \in Terms_T$ *for all* $f \in FSym$ *with* $\alpha(f) = (T_1, \ldots, T_n) \times T$ *and* $t_i \in Terms_{T'_i}$ *with* $T'_i \sqsubseteq T_i$
- $\backslash if(\varphi)\,\backslash then(t_1)\,\backslash else(t_2) \in Terms_T$ *for all formulas* φ *and all* $t_1 \in Terms_{T_1}, t_2 \in Terms_{T_2}$ *with* $T_1 \sqcup T_2 = T$ *(where* $T_1 \sqcup T_2$ *denotes the smallest common supertype of* T_1, T_2*).*
- $\{\mathcal{U}\}t \in Terms_T$ *for all* $t \in Terms_T$ *and all updates* $\mathcal{U}$

Definition 2.4 (Formulas) *The set* Fml *of* Java DL *formulas is the smallest set such that*

- $P(t_1, \ldots, t_n) \in Fml$ *for all* $P \in PSym$ *with* $\alpha(p) = (T_1, \ldots, T_n)$ *and* $t_i \in Terms_{T'_i}$ *with* $T'_i \sqsubseteq T_i$
- Fml *is closed under first order connectives*
- *for all* $\varphi \in Fml$ *and* $x \in VSym$ *with* $\alpha(x) = T$ *then* $\forall\, T\ x; \varphi \in Fml$ *and* $\exists\, T\ x; \varphi \in Fml$
- $\backslash if(\varphi)\,\backslash then(\psi_1)\,\backslash else(\psi_2) \in Fml$ *for all* $\varphi, \psi_1, \psi_2 \in Fml$

[2] We ignore, for instance, $\backslash ifEx$ terms here since they do not occur in the remainder of this work.

- *$\{\mathcal{U}\}\phi \in Fml$ for all $\phi \in Fml$ and all updates $\mathcal{U}$*
- *$[p]\phi \in Fml$ and $\langle p\rangle\phi \in Fml$ for all* Java DL *programs p and all formulas $\phi \in Fml$*

For the sake of simplicity the type of quantified variables is usually omitted in the following.

Definition 2.5 (Programs in Java DL) *A* Java DL *program is a sequence of legal* Java DL *statements where a legal* Java DL *statement is either*

- *a* Java *statement or*
- *a method frame or method body statement (see Section 2.3.6)*

Program locations are also represented as flexible Java DL terms. The function symbols corresponding to local variables, attributes and the array access operator are called *location function symbols*.

Definition 2.6 (Location Function Symbol) *A location function symbol is either*

- *a program variable v (arity 0) representing a local variable or a static attribute or*
- *an instance attribute $a@(T)$ (arity 1, $\alpha(a@(T)) = T \times T_{a@(T)}$ where $T_{a@(T)}$ is the static type of attribute $a@(T)$) declared in T or*
- *the array access operator $[]$ (arity 2, $\alpha([]) = (T[], int) \times T$ where T is the static element type of the array)*

The set $LocSym$ (with $LocSym \subseteq FSym_{nr}$) denotes the set of all location function symbols.

For technical reasons we define another class of flexible function symbols used in the calculus for memorizing the values of locations certain program states (such as the pre-state of method invocations). Virtual locations may not occur in programs but on the left-hand-side of updates.

Definition 2.7 (Virtual Location Function Symbol) *A function symbol $f \in FSym_{nr}$ is a virtual location function symbol if and only if*

- *f is not a location function symbol and*
- *f is not a location dependent symbol (see Definition 2.9)*

The set of all virtual location function symbols is denoted with $VLocSym$.

A state in a Java DL Kripke structure is uniquely determined by the evaluation of all function symbols $f \in LocSym \cup VLocSym$. Nevertheless there is a third kind of flexible symbols, called *location dependent symbols* [Bubel, 2007], whose values do not determine the state explicitly but are “affected” by parts of the state, namely by the value of certain locations they *depend on*. Location dependent symbols cannot occur in programs or on the left-hand side of updates, thus their value cannot be changed explicitly. A location dependent symbol may implicitly depend on the set of all locations. In this case there are no restrictions on its interpretation. It is, however, also possible to constrain the set of locations a location dependent symbol depends on explicitly using by location descriptors, a concept also introduced in [Bubel, 2007] for describing sets of locations. For this work only location dependent symbols with explicit dependencies are relevant.

Definition 2.8 (Location Descriptor) *A* location descriptor *has the form*

$$*$$

or

$$(\mathit{for}\, x_1 \ldots x_n;\ \mathit{if}(\varphi)\ f(t_1, \ldots, t_m))$$

where

- $x_1 \ldots x_n$ *are logic variables bound in* φ *and* $t_1, \ldots, t_m$
- φ *is an arbitrary formula*
- f *is a location or a virtual location function symbol*
- $t_1, \ldots, t_m$ *are arbitrary terms complying with the signature* $\alpha(f)$*.*
- *No free variables occur in a location descriptor. Thus the only free variables that can occur in* φ *and* $t_1 \ldots t_m$ *are* $x_1 \ldots x_n$ *which are bound by the location descriptor.*

In cases no variables are bound in φ and $t_1, \ldots, t_m$ we write

$$\mathit{if}(\varphi)\ f(t_1, \ldots, t_m)$$

If, in addition, φ is identical to true we just write $f(t_1, \ldots, t_m)$. The set of locations described by a location descriptor in a certain state is referred to as its extension. The location descriptor $*$ represents the entire heap.

Definition 2.9 (Location Dependent Symbol with explicit Dependencies) *Let*

$$\mathit{Locs} := ld_1, \ldots, ld_n$$

be a list of location descriptors then the flexible predicate symbol

$$P[\mathit{Locs}]$$

is called a location dependent predicate symbol and flexible function symbol

$$f[\mathit{Locs}]$$

is called a location dependent function symbol.

Semantical differences between the original definition [Bubel, 2007] of location dependent symbols and the one used in this work are pointed out in Section 2.2.2.

Definition 2.10 (Updates) *Let* f *be a location or a virtual location function symbol and* $t_1, \ldots, t_n$ *and* v *terms such that* $f(t_1, \ldots, t_n) \in \mathit{Terms}_{T_1}$ *and* $v \in \mathit{Terms}_{T_2}$ *with* $T_2 \sqsubseteq T_1$*. Let further* $\mathcal{U}_1$ *and* $\mathcal{U}_2$ *be updates,* x *a logic variable and Locs a list of location descriptors then*

- *skip is an update (empty update)*
- $f(t_1, \ldots, t_n) := v$ *is an update (elementary update)*
- $\mathcal{U}_1\ ;\ \mathcal{U}_2$ *is an update (sequential composition of update)*

- $\mathcal{U}_1 \,||\, \mathcal{U}_2$ *is an update (parallel composition of update)*
- *if*$(\phi)\, \mathcal{U}_1$ *is an update (conditional update)*
- *for* $T\ x$; *if*$(\phi)\, \mathcal{U}_1$ *is an update (quantified conditional update), x is bound in ϕ and* $\mathcal{U}_1$
- $*_i^{Locs}$ *is an update where $i \in \mathbb{N}$ (anonymising update)*
- $*_i$ *is an update where $i \in \mathbb{N}$ (anonymous update)*

In the following we may omit the type of the logic variable bound in a quantified update or write

$$\text{for } x_1, \ldots, x_n;\ \text{if}(\phi)\, \mathcal{U}_1$$

as syntactic sugar for

$$\text{for } x_1;\ \text{if}(true)\ \text{for } x_2; \ldots \text{for } x_n;\ \text{if}(\phi)\, \mathcal{U}_1$$

2.2 Semantics

The semantic domain for Java DL are, as for modal logics in general, *Kripke structures*. For a more convenient semantical definition of anonymous updates we extend our notion of a dynamic logic Kripke structure in the following a bit by introducing a mapping from anonymising updates to "reference states".

Definition 2.11 (Java DL Kripke Structure) *Let Σ be a* Java DL *signature for a type hierarchy $(\mathcal{T}, \sqsubseteq)$. A* Java DL *Kripke structure for Σ and the given type hierarchy is a tuple $(\mathcal{M}, \mathcal{S}, *, \rho)$ such that*

- *$\mathcal{M} := (\mathcal{D}, I)$ is a partial first order structure (with domain $\mathcal{D}$ and interpretation I) fixing the interpretation of $FSym_r$ and $PSym_r$ (of the given signature) where $\mathcal{D} := (\mathbf{U}, \delta, \preceq)$ consisting of*
  - *the universe* $\mathbf{U}$
  - *a typing function $\delta : \mathbf{U} \to \mathcal{T}$ assigning each $v \in \mathbf{U}$ its exact (dynamic) type. We define:*

    $$I(T) := \{v \in \mathbf{U} \mid \delta(v) \sqsubseteq T\}$$
  - *A well-ordering $\preceq$ defined on* $\mathbf{U}$
- *$\mathcal{S}$ is a set of states where each state $s \in \mathcal{S}$ is a first order structure refining $\mathcal{M}$ by fixing also the interpretation of $FSym_{nr}$ and $PSym_{nr}$.*
- *The mapping $* : \mathbb{N} \to \mathcal{S}$*
- *The mapping ρ assigning to each program p the state transition relation $\rho(p) \subseteq \mathcal{S} \times \mathcal{S}$*

Note that the domain being fixed in $\mathcal{M}$ entails that it is the same in all states of a Kripke structure (*constant domain assumption*, see also Section 2.3.3). The partial structure $\mathcal{M}$ is called the *Kripke seed* .

The interpretation of terms and formulas is defined largely in a standard way (for details refer to [Beckert et al., 2007]) so we only focus on the semantics of updates and location dependent symbols in the following. Validity and satisfiability of formulas are also defined as usual for a dynamic logic.

Definition 2.12 (Satisfiability and Validity of Java DL Formulas) *Let ϕ be a* Java DL *formula. Then ϕ is satisfiable if and only if there is a Kripke structure $(\mathcal{M}, \mathcal{S}, *, \rho)$ (for the given program context), a state $s \in \mathcal{S}$ and a variable assignment β such that*

$$s, \beta \models \phi$$

The formula ϕ is valid if and only if

$$\mathcal{K} \models \phi$$

for all (with the regard to the given program context) Kripke structures $\mathcal{K}$. We write $\mathcal{K} \models \phi$ for $\mathcal{K} := (\mathcal{M}, \mathcal{S}, *, \rho)$ if and only if for all states $s \in \mathcal{S}$ and all variable assignments β

$$s, \beta \models \phi$$

2.2.1 Updates

For talking about the semantics of updates we first define the notion of a semantic location, which is a semantic entity independent of a syntactical representation of the location.

Definition 2.13 (Semantic Location) *Let f be a location (or virtual location) function symbol with $(T_1, \ldots, T_n) \times T$ and $\bar{v} := v_1, \ldots, v_n$ a vector of elements of $\mathbf{U}$ with $v_i \in I(T_i)$. Then the tuple*

$$(f, \bar{v})$$

is called a semantic location.

Analogous to this definition of semantic locations we define a semantic update as a mapping from semantic locations to values (elements of $\mathbf{U}$).

Definition 2.14 (Semantic Update) *A* semantic update *is a set of pairs (l, v), where l is a semantic location and $v \in \mathbf{U}$. We call a semantic update U consistent if for no location l there are $(l, v_1) \in U$ and $(l, v_2) \in U$ such that $v_1 \neq v_2$. A semantic update U can be seen as a function on states, where for each state s the state $U(s)$ is partly defined*[3] *by the semantic update fixing the interpretation of location (and virtual location) function symbols (by this $U(s)$ is uniquely determined)*

$$U(s)(f)(\bar{v}) := \begin{cases} v & \textit{if } ((f, \bar{v}), v) \in U \\ s(f)(\bar{v}) & \textit{otherwise} \end{cases}$$

for all location function symbols f and legal tuple of values $\bar{v}$.

[3] One could also say partly redefined relative to state s, as state s is preserved except for the locations occurring in U.

Now we can define how updates relate to semantic updates.

Definition 2.15 (Evaluation of Updates) *Let $\mathcal{K} := (\mathcal{M}, \mathcal{S}, *, \rho)$ be a* Java DL *Kripke structure, s a state with $s \in \mathcal{S}$ and β a variable assignment. An update $\mathcal{U}$ is evaluated to a consistent semantic update $val_{s,\beta}(\mathcal{U})$ in the following way:*

- $val_{s,\beta}(skip) := \emptyset$
- $val_{s,\beta}(f(t_1, \ldots, t_n) := t) := \{((f, (val_{s,\beta}(t_1), \ldots, val_{s,\beta}(t_n))), val_{s,\beta}(t))\}$
- $val_{s,\beta}(if(\phi)\, \mathcal{U}) := \begin{cases} val_{s,\beta}(\mathcal{U}) & if\ s, \beta \models \phi \\ \emptyset & otherwise \end{cases}$
- $val_{s,\beta}(for\, T\ x;\ if(\phi)\, \mathcal{U}) := win\big(\bigcup_{v \in I(T):\ s,\beta_x^v \models \phi} \{val_{s,\beta_x^v}(\mathcal{U})\}\big)$

  The function win ensures that the resulting update is consistent (meaning each location is mapped to at most one value). This is achieved by utilizing the well-ordering $\preceq$ on $\mathbf{U}$ and choosing the smallest value v in case a clash occurs (for details refer to [Rümmer, 2006]).
- $val_{s,\beta}(\mathcal{U}_1 \,||\, \mathcal{U}_2) := (U_1 \cup U_2) \setminus C$

  where $U_1 := val_{s,\beta}(\mathcal{U}_1)$, $U_2 := val_{s,\beta}(\mathcal{U}_2)$, and where
  $C := \{((f, \bar{v}), v) \in U_1 \mid ((f, \bar{v}), v') \in U_2$ for some $v' \neq v\}$
- $val_{s,\beta}(\mathcal{U}_1\ ;\ \mathcal{U}_2) := (U_1 \cup U_2) \setminus C$

  where $U_1 := val_{s,\beta}(\mathcal{U}_1)$, $U_2 := val_{U_1(s),\beta}(\mathcal{U}_2)$, and where
  $C := \{((f, \bar{v}), v) \in U_1 \mid ((f, \bar{v}), v') \in U_2$ for some $v' \neq v\}$
- $val_{s,\beta}(*_i) := \{((f, \bar{v}), *(i)(f)(v_1, \ldots, v_n)) |$ *f is location function symbol with* $\alpha(f) = (T_1, \ldots, T_n) \times T$ *and $t_i \in Terms_{T'_i}$ with $T'_i \sqsubseteq T_i$*$\}$
- $val_{s,\beta}(*_i^{Locs}) := \{((f, \bar{v}), *(i)(f)(\bar{v})) |\ (f, \bar{v}) \in val_{s,\beta}(Locs)\}$

Note, that an anonymous update $*_i$ only anonymises locations (*not* virtual locations).

2.2.2 Location Descriptors and Location Dependent Symbols

In [Bubel, 2007] the concept of location descriptors (see Definition 2.8) is introduced for describing sets of locations, which is for instance needed to express change information for methods or their data dependencies. In the following the key ideas behind location descriptors are summarized and adapted and extended to our purposes when necessary.

Definition 2.16 (Location Descriptor Extension) *Let $\mathcal{K} := (\mathcal{M}, \mathcal{S}, *, \rho)$ be a* Java DL *Kripke structure, $ld := (for\ \bar{x};\ if(\varphi)\ f(\bar{t}))$ a location descriptor and $s \in \mathcal{S}$. Then $val_s(ld)$ denotes the set of locations ld evaluates to in state s:*

$$val_s(ld) := \{(f, (val_{s,\beta}(\bar{t}))) | s, \beta \models \varphi\}$$

The extension of a list $Locs := ld_1, \ldots, ld_n$ of location descriptors is the union of the extension sets over all location descriptors contained in Locs:

$$val_s(Locs) := \bigcup_{i \in \{1,\ldots,n\}} val_s(ld_i)$$

The extension of the special location descriptor $*$ representing the set of all heap locations is given by:

$$val_s(*) := \left\{ (f, (a_1, \ldots, a_n)) \;\middle|\; \begin{array}{l} \alpha(f) = (T_1, \ldots, T_n) \times T \textit{ and} \\ \delta(a_i) \sqsubseteq T_i \textit{ and} \\ \left(\begin{array}{l} f \textit{ is an attribute or} \\ f = [] \end{array} \right) \end{array} \right\}$$

Remark 2.17 (Anonymising Updates for Location Descriptors) *As stated in Definition 2.15, an anonymising update $*_i^{Locs}$ maps each location $l \in val_s(Locs)$ (for an arbitrary state s) to the value l evaluates to in the reference state $*(i)$. Thus the anonymising update*

$$*_i^{ld}$$

with $ld := (\textit{for } \bar{x};\ \textit{if}(\varphi)\ f(\bar{t}))$ where f is a location function symbol is obviously equivalent to the update

$$\{\textit{for } \bar{x};\ \textit{if}(\varphi)\ f(\bar{t}) := (\{*_i\}f)(\bar{t})\} \tag{2.1}$$

In [Beckert et al., 2007] a location descriptor (a terminology, however, not used in the cited work) $ld := (\textit{for } \bar{x};\ \textit{if}(\varphi)\ f(\bar{t}))$ is anonymised by the update

$$\{\textit{for } \bar{x};\ \textit{if}(\varphi)\ f(\bar{t}) := f'(\bar{t})\} \tag{2.2}$$

where f' is a fresh rigid function symbol not yet occurring in the regarded sequent in which the update occurs. The two approaches of defining anonymising updates are "interchangeable" as for each Kripke structure $\mathcal{K} := (\mathcal{M}, \mathcal{S}, *, \rho)$ and for each rigid function symbol f' and each location function symbol f there is Kripke structure and $\mathcal{K}' := (\mathcal{M}', \mathcal{S}', *', \rho')$ (with $\mathcal{M}' := (\mathcal{D}', I')$) such that

$$*(i)(f) = I'(f')$$

*and vice versa. The semantics for anonymous updates differ however since we restrict the effect of anonymous updates to (*non-virtual*) location function symbols.*

Thus a formula free of anonymous updates (but possibly containing anonymising updates) is satisfiable (resp. valid) applying the definition in [Beckert et al., 2007] if and only if it is satisfiable (valid) applying our definition for anonymising updates.

Example 2.18 (Location Descriptors)

- *The location descriptor $ld := (\textit{for int } x;\ \textit{if}(0 \leq x \wedge x < arr.length)\ arr[x])$ contains all array slots the array arr possesses in the state ld is evaluated in.*
- *$\textit{if}(o \neq$ **`null`**$)\ o.a$ contains the location o.a if evaluated in states s in which o is not **`null`**. In all other states the extension of this location descriptor is the empty set.*

In [Bubel et al., 2008a] location dependent symbols have been introduced. The interpretation of a symbol $P[ld]$ depending on a location descriptor ld has to coincide in each two states s_1, s_2 that share the same values for the locations described by ld. This circumstance is formalized by definition 2.19 and 2.23.

Definition 2.19 (The Relation $\approx_{ld}$) *Let $\mathcal{K} := (\mathcal{M}, \mathcal{S}, *, \rho)$ be a* Java DL *Kripke structure and $ld := (\textit{for } \bar{x};\ \textit{if}(\varphi)\ f(\bar{t}))$ a location descriptor. For two states $s_1, s_2 \in \mathcal{S}$ the relation $s_1 \approx_{ld} s_2$ holds if and only if for every variable assignment β*

- $s_1, \beta \models \varphi$ *if and only if* $s_2, \beta \models \varphi$ *and*
- $val_{s_1,\beta}(f(\bar{t})) = val_{s_2,\beta}(f(\bar{t}))$ *if* $val_{s_1,\beta} \models \varphi$

Let Locs be a list of location descriptors then $s_1 \approx_{Locs} s_2$ *holds if and only if* $s_1 \approx_{ld} s_2$ *for all* $ld \in Locs$.

We write $s_1 \approx_{*} s_2$ *if the heap is identical in* s_1 *and* s_2 *meaning* $val_{s_1}(f) = val_{s_2}(f)$ *for all location function symbols* f.

Note, that there are subtle differences in the definition of $\approx_{ld}$ in this work compared to how the respective relation is defined in [Bubel et al., 2008a] (we call the *location descriptor equivalence* defined in that work $\approx'_{ld}$ to distinguish it from our definition). In particular, we do not require the extension of ld to be equal in s_1 and s_2 if $s_1 \approx_{ld} s_2$ holds. Let us, for instance, consider the following simple location descriptor: $ld := \{o.a\}$. For $s_1 \approx_{ld} s_2$ it is sufficient that $val_{s_1}(o.a) = val_{s_2}(o.a)$. In contrast, the definition of $\approx'_{ld}$ from [Bubel et al., 2008a] would also require that $val_{s_1}(o) = val_{s_2}(o)$ in order for the extensions of ld to be equal in s_1 and s_2. In this respect our definition is more liberal in another one, however, it is more restrictive as illustrated with this method specification:

— Java + JML —

```
/*@ requires o!=null && u!=null;
  @ depends o.i, u.i;
  @*/
public int div(MyInteger o, MyInteger u){
  return o.i/u.i;
}
```

— Java + JML —

The *depends* clause of the above specification is given by the list of location descriptors $Dep := \{o.i, u.i\}$. Let us consider two states s_1 and s_2 with

$$val_{s_1}(o) = obj_1,\ val_{s_1}(u) = obj_2,\ val_{s_1}(o.i) = i_1,\ val_{s_1}(u.i) = i_2 \qquad (2.3)$$
$$val_{s_2}(o) = obj_2,\ val_{s_2}(u) = obj_1,\ val_{s_2}(o.i) = i_2,\ val_{s_2}(u.i) = i_1 \qquad (2.4)$$

where $obj_1 \neq obj_2$ (the interpretations of o and u are just swapped in state s_2 in comparison to s_1) and $i_1 \neq i_2$. The extension of Dep is therefore the same in both s_1 and s_2, namely

$$val_{s_1}(Dep) = val_{s_2}(Dep) = \{(i, (obj_1)), (i, (obj_2))\}$$

Together with

$$s_1(i)(obj_1) = val_{s_1}(o.i) = i_1 = val_{s_2}(u.i) = s_2(i)(obj_1)$$

and

$$s_1(i)(obj_2) = val_{s_1}(u.i) = i_2 = val_{s_2}(o.i) = s_2(i)(obj_2)$$

we get $s_1 \approx'_{Dep} s_2$. This is, however, contrary to our intuition since i_1/i_2 (as computed by `div` when executed in s_1) is obviously not always the same as i_2/i_1 (as computed by `div` when executed in s_2) and thus s_1 and s_2 should not be considered equivalent with respect to `div`'s *depends* clause Dep. This consideration is taken into account by definition 2.19 which requires that for two states to be equivalent with respect to Dep both states must be equivalent with respect to each $ld \in Dep$. Therefore, $s_1 \not\approx_{Dep} s_2$ since $s_1 \not\approx_{o.a} s_2$ (due to $val_{s_1}(o.a) = i_1 \neq i_2 = val_{s_2}(o.a)$) and $s_1 \not\approx_{u.a} s_2$ (due to $val_{s_1}(u.a) = i_2 \neq i_1 = val_{s_2}(u.a)$).

Remark 2.20 (Mimicking the Semantics of $\approx'_{ld}$ (to some degree)) *In certain cases*[4] *it can still be necessary to identify states s, t in which the extension of a location descriptor ld is identical. This is not entailed by $s \approx_{ld} t$ as it was just elaborated on. This aspect of $\approx'_{ld}$ can, however, easily be mimicked: Let $ld := (\mathit{for}\ \bar{x};\ \mathit{if}(\varphi)\ f(\bar{t}))$ be a location descriptor, where $\bar{x} := x_1 \ldots x_n$ and $\bar{t} := t_1, \ldots, t_m$. We define*

$$ld_{ext} := (\mathit{for}\ \bar{x}, \bar{y};\ \mathit{if}(\varphi \wedge \bigwedge_{i \in \{1,\ldots,m\}} y_i \doteq t_i)\ f(\bar{t}))$$

where $\bar{y} := y_1, \ldots, y_m$ and $\bar{y}$ and $\bar{x}$ are disjoint. For each two states s and t we get

$$s \approx_{ld_{ext}} t \ \ \mathit{implies}\ \ s \approx'_{ld} t$$

as stated by Lemma 2.21 and proven in Proof 1.

Accordingly, for a list $Locs := ld_1, \ldots, ld_n$ of location descriptors we define

$$Locs_{ext} := ld_{1_{ext}}, \ldots, ld_{n_{ext}}$$

Lemma 2.21 *Let Locs be a list of location descriptors. Then for every Kripke structure $\mathcal{K} := (\mathcal{M}, \mathcal{S}, *, \rho)$ and any two states $s_1, s_2 \in \mathcal{S}$*

$$s_1 \approx_{Locs_{ext}} s_2 \tag{2.5}$$

entails

$$s_1 \approx_{Locs} s_2 \tag{2.6}$$

and

$$s_1 \approx'_{Locs} s_2 \tag{2.7}$$

Proof 1 (of Lemma 2.21) *We first show that equation (2.6) and then that equation (2.7) follows directly from definition 2.19.*

Equation (2.6): We assume that there are states s_1, s_2 such that

$$s_1 \approx_{Locs_{ext}} s_2 \tag{2.8}$$

and

$$s_1 \not\approx_{Locs} s_2 \tag{2.9}$$

Then according to definition 2.19 there is a location descriptor $ld := (\mathit{for}\ \bar{x};\ \mathit{if}(\varphi)\ f(\bar{t}))$ and a variable assignment β such that either

- *$s_1, \beta \models \varphi$ if and only if $s_2, \beta \models \varphi$ does not hold. Wlog. we assume that $s_1, \beta \models \varphi$ and $s_2, \beta \not\models \varphi$. Then we could, however, find a variable assignment β' coinciding with β in the assignments for the variables $\bar{x}$ such that*

$$s_1, \beta' \models \bigwedge_{i \in \{1,\ldots,m\}} y_i \doteq t_i$$

[4] In this work an example for this is given by the PO *RespectsDep*(p, ct) in Section 9.2.1.

and therefore

$$s_1, \beta' \models \varphi \wedge \bigwedge_{i \in \{1,\ldots,m\}} y_i \doteq t_i$$

and

$$s_2, \beta' \not\models \varphi \wedge \bigwedge_{i \in \{1,\ldots,m\}} y_i \doteq t_i$$

which poses a contradiction to the assumption (2.8).

- $val_{s_1,\beta}(f(\bar{t})) \neq val_{s_2,\beta}(f(\bar{t}))$ *and* $val_{s_1,\beta} \models \varphi$*. Then we can obviously find a variable assignment* β' *coinciding with* β *in the assignments to* $\bar{x}$ *such that*

  $$s_1, \beta' \models \varphi \wedge \bigwedge_{i \in \{1,\ldots,m\}} y_i \doteq t_i$$

  due to (2.8) we get

  $$val_{s_1,\beta'}(f(\bar{t})) = val_{s_2,\beta'}(f(\bar{t})) \tag{2.10}$$

  since β' *coincides with* β *in the assignments to the variables* $\bar{x}$ *which are the only free variables in* $\bar{t}$

  $$val_{s_1,\beta'}(f(\bar{t})) = val_{s_1,\beta}(f(\bar{t})) \tag{2.11}$$

  $$val_{s_2,\beta'}(f(\bar{t})) = val_{s_2,\beta}(f(\bar{t})) \tag{2.12}$$

  and thus together with equation (2.10)

  $$val_{s_1,\beta}(f(\bar{t})) = val_{s_2,\beta}(f(\bar{t}))$$

  which contradicts the initial assumption that $val_{s_1,\beta}(f(\bar{t})) \neq val_{s_2,\beta}(f(\bar{t}))$ *holds.*

Equation (2.7): Definition 2.19 requires for $s_1 \approx_{Locs_{ext}} s_2$ *to hold that for every location descriptor* $ld_{ext} := ($*for* $\bar{x}, \bar{y}$*; if*$(\varphi \wedge \bigwedge_{i \in \{1,\ldots,m\}} y_i \doteq t_i)\ f(\bar{t}))$ *contained in* $Locs_{ext}$ *and every variable assignment* β

$$s_1, \beta \models \varphi \wedge \bigwedge_{i \in \{1,\ldots,m\}} y_i \doteq t_i \quad \textit{iff.} \quad s_2, \beta \models \varphi \wedge \bigwedge_{i \in \{1,\ldots,m\}} y_i \doteq t_i$$

which entails that $val_{s_1,\beta}(t_i) = val_{s_2,\beta}(t_i)$ *(for* $i \in 1, \ldots, n$*) for every location*

$$(f, (val_{s_1,\beta}(t_1), \ldots, val_{s_1,\beta}(t_n))) \in val_{s_1}(ld_{ext})$$

and thus

$$(f, (val_{s_1,\beta}(t_1), \ldots, val_{s_1,\beta}(t_n))) = (f, (val_{s_2,\beta}(t_1), \ldots, val_{s_2,\beta}(t_n))) \in val_{s_2}(ld_{ext})$$

which entails $val_{s_1}(ld_{ext}) \subseteq val_{s_2}(ld_{ext})$ *and (since for symmetry reasons also* $val_{s_2}(ld_{ext}) \subseteq val_{s_1}(ld_{ext})$ *holds):*

$$val_{s_1}(ld_{ext}) = val_{s_2}(ld_{ext}) \tag{2.13}$$

From equation (2.6) we get that in addition

$$s_1(f)(val_{s_1,\beta}(t_1), \ldots, val_{s_1,\beta}(t_n)) = s_2(f)(val_{s_2,\beta}(t_1), \ldots, val_{s_2,\beta}(t_n))$$

for every location

$$(f, (val_{s_1,\beta}(t_1), \ldots, val_{s_1,\beta}(t_n))) \in val_{s_1}(Locs)$$

and thus

$$s_1 \approx'_{Locs} s_2$$

Example 2.22 *Let us consider the already previously used set of location descriptors* $Locs := o.i@(MyInteger), u.i@(MyInteger)$ *then*

$$Locs_{ext} = (for\, y_1;\ if(y_1 \doteq o)\ o.i), (for\, y_2;\ if(y_2 \doteq u)\ u.i)$$

Since $s_1 \approx_{Locs_{Ext}} s_2$ *entails that for every variable assignment* β

$$s_1, \beta \models y_1 \doteq o \text{ iff. } s_2, \beta \models y_1 \doteq o$$

and thus (since $val_{s_1,\beta}(y_1) = \beta(y_1) = val_{s_2,\beta}(y_1)$*) we get*

$$(i, val_{s_1,\beta}(o)) = (i, val_{s_2,\beta}(o))$$

As this holds analogously for the location descriptor u.i, the extensions of Locs are the same in s_1 *and* s_2*:*

$$val_{s_1,\beta}(Locs) = val_{s_2,\beta}(Locs)$$

Due to $s_1 \approx_{Locs_{Ext}} s_2$ *also* $val_{s_1}(o.i) = val_{s_2}(o.i)$ *and* $val_{s_1}(u.i) = val_{s_2}(u.i)$*, therefore:*

$$s_1 \approx'_{Locs} s_2$$

Intuitively the semantics of a location dependent symbol requires, that it is interpreted identically in two states s and t of a Kripke structure $\mathcal{K}$ if the evaluation of the locations ($Locs$) it depends on is identical in both states ($s \approx_{Locs} t$). Note, that this does not mean that a location dependent symbol is an interpreted symbol or that its value is fixed by the evaluation of the locations $Locs$ it depends on for all Kripke structures: There are no restrictions on how a location dependent symbol is interpreted in two states s and s' contained in two different Kripke structures $\mathcal{K}$ and $\mathcal{K}'$ irrespective of how the locations it depends on are interpreted in these two states.

Definition 2.23 (Semantics of Location Dependent Symbols) *Let LDs be a list of location descriptors,* $\mathcal{K} := (\mathcal{M}, \mathcal{S}, *, \rho)$ *a* Java DL *Kripke structure and* $P[LDs]$ *(respectively* $f[LDs]$*) a predicate (function) symbol depending on LDs. Then for any two states* $s, t \in \mathcal{S}$ *with* $s \approx_{LDs} t$ *the interpretation of* $P[LDs]$ *(*$f[LDs]$*) in s and t coincides.*

In certain cases, such as for defining proof obligations for the correctness of assignable or *depends* clauses (see Section 9.2.1 for details), it is necessary to memorize what locations a location descriptor denotes in a certain state (which is the extension of the location descriptor). Technically we realize this by means of updates storing the relevant information in virtual locations.

Definition 2.24 (The Update $\mathcal{D}@pre(ld)$ **and the Location Descriptor** $ld^{@pre}$**)** *Let*

$$ld := (for\, x_1 \ldots x_n;\ if(\varphi)\ f(t_1, \ldots, t_m))$$

be a location descriptor. We define

$$\mathcal{D}@pre(ld) := \{\mathcal{U}_\varphi \,||\, \mathcal{U}_{t_1} \,||\ldots||\mathcal{U}_{t_m}\}$$

with

$$\begin{array}{lcl} \mathcal{U}_\varphi & := & for\, x_1 \ldots x_n;\ if(\varphi)\ b_\varphi^{@pre}(x_1, \ldots, x_n) := TRUE \\ \mathcal{U}_{t_i} & := & for\, x_1 \ldots x_n;\ if(true)\ t_i^{@pre}(x_1, \ldots, x_n) := t_i \quad for\, 1 \le i \le m \end{array}$$

and $b_\varphi^{@pre}, t_i^{@pre}$ being virtual location function symbols of appropriate type.
We define the location descriptor $ld^{@pre}$ as:

$$ld^{@pre} := (\mathit{for}\,\bar{x};\ \mathit{if}(b_\varphi^{@pre}(\bar{x}) \doteq \mathit{TRUE})\ f(t_1^{@pre}(\bar{x}), \ldots, t_m^{@pre}(\bar{x})))$$

where $\bar{x} := x_1, \ldots, x_n$.

Since, except for the top level location function f, only virtual location function symbols and also no location dependent symbols occur in the $ld^{@pre}$ location descriptor, its evaluation is not affected by changes of the heap, which is stated by the following theorem.

Theorem 1 (Extension of $ld^{@pre}$) *Let ld be a location descriptor and $\mathcal{D}@pre(ld)$ and $ld^{@pre}$ be defined as in definition 2.24. Then the following holds for every Kripke structure $\mathcal{K} := (\mathcal{M}, \mathcal{S}, *, \rho)$ every $s \in \mathcal{S}$, $i \in \mathbb{N}$ and every program p:*

$$val_s(ld) = val_t(*_i)(t)(ld^{@pre}) \tag{2.14}$$
$$val_s(ld) = val_{\rho(p)(t)}(ld^{@pre}) \tag{2.15}$$

where $t := val_s(\mathcal{D}@pre(ld))(s)$.

Example 2.25 *Let us consider the location descriptor $ld := o.attr$ then*

$$\mathcal{D}@pre(ld) = o^{@pre} := o \tag{2.16}$$
$$ld^{@pre} = o^{@pre}.attr \tag{2.17}$$

obviously $(attr, val_s(o)) = (attr, val_t(o^{@pre}))$ (for t as defined in Theorem 1) and thus also equations (2.14) and (2.15) hold since the value of the virtual location $o^{@pre}$ is not changed by update $*_i$ or by a program p (see definitions 2.7 and 2.15).

There are also cases in which we do not only need to memorize the locations described by a location descriptor in a certain state but also their values (examples for this can again be found in Section 9.2.1). For this purpose we define the update $\mathcal{D}@pre'(ld)$ and the corresponding location descriptor $ld^{@pre'}$.

Definition 2.26 (The Update $\mathcal{D}@pre'(ld)$ and the Location Descriptor $ld^{@pre'}$) *Let*

$$ld := (\mathit{for}\, x_1 \ldots x_n;\ \mathit{if}(\varphi)\ f(t_1, \ldots, t_m))$$

be a location. We define

$$\mathcal{D}@pre'(ld) := \{\mathcal{D}@pre(ld) || \mathit{for}\,\bar{y};\ \mathit{if}(\mathit{true})\ f^{@pre}(\bar{y}) := f(\bar{y})\}$$

where $f^{@pre}$ is a virtual location function symbol and $\bar{y} := y_1, \ldots, y_m$.
Accordingly, we define the location descriptor $ld^{@pre'}$ as:

$$ld^{@pre'} := (\mathit{for}\,\bar{x};\ \mathit{if}(b_\varphi^{@pre}(\bar{x}) \doteq \mathit{TRUE})\ f^{@pre}(t_1^{@pre}(\bar{x}), \ldots, t_m^{@pre}(\bar{x})))$$

where $\bar{x} := x_1, \ldots, x_n$.

As we can easily see, the information “stored” by $\mathcal{D}@pre'(ld)$ subsumes the information stored by $\mathcal{D}@pre(ld)$. We can therefore also safely use $\mathcal{D}@pre'(ld)$ to initialise the virtual locations needed for defining $ld^{@pre}$. Since $ld^{@pre'}$ is not heap dependent, its extension as well as the values of the locations contained in its extension are not affected by changes of the heap. This is expressed by the following theorem.

Theorem 2 (Extension of $ld^{@pre'}$) *Let ld be a location descriptor and $\mathcal{D}@pre'(ld)$, $ld^{@pre'}$ be defined as in definition 2.26. Then the following holds for every Kripke structure $\mathcal{K} := (\mathcal{M}, \mathcal{S}, *, \rho)$ and every $s, r \in \mathcal{S}$ such that there is an $i \in \mathbb{N}$ or a program p with $r = \rho(p)(t)$ or $r = val_t(*_i)(t)$ (where $t := val_s(\mathcal{D}@pre'(ld))(s)$):*

$$\begin{aligned} (f, (\bar{a})) \in val_s(ld) \text{ iff. } (f^{@pre}, (\bar{a})) \in val_r(ld^{@pre'}) \\ s(f)(\bar{a}) = r(f^{@pre})(\bar{a}) \quad \text{f.a. } (f, (\bar{a})) \in val_s(ld) \end{aligned}$$

2.3 Calculus

The calculus we employ is a sequent calculus for Java DL which uses symbolic execution [King, 1976] for evaluating programs. In this Section we shortly review parts of the calculus that are of particular relevance for this work. Special emphasis is placed on newly made extensions to the calculus, such as the handling of inner classes and comprehensions, which were a prerequisite for parts of this work.

2.3.1 The Taclet Language

Calculus rules in the KeY system are written in the *taclet* language. This section gives a quick overview of the taclet language as far as needed to understand the remainder of this work and introduces a textbook-style notation[5] for taclets. A comprehensive overview of the taclet language can be found in [Beckert et al., 2004, Giese, 2004] or in Chapter 4 of [Beckert et al., 2007].

Introductory Examples

find* and *replacewith Let us consider the propositional logic rule treating implications occurring as top-level formulas in the succedent:

$$\mathsf{impRight}\ \frac{\Gamma,\ \phi \Rightarrow \psi,\ \Delta}{\Gamma \Rightarrow \phi \to \psi,\ \Delta}$$

This rule can be formulated in the taclet language as follows:

— Taclet —

```
impRight{
  \find(==> phi -> psi)
  \replacewith(phi ==> psi)
}
```

— Taclet —

[5] This newly defined notation does not provide the full expressiveness of the taclet language.

The entities `phi` and `psi` occurring in the above taclet are so-called schema variables which must be declared beforehand. They are typed and can only match logic entities of appropriate type. The *find* clause describes a pattern (or schema) for sequents, terms or formulas to which the rule is applicable, in this case sequents containing a formula of the form $\phi \rightarrow \psi$ in the succedent. The *replacewith* clause specifies a schema for the sequent (term, formula) used to replace the sequent matched in the *find* part. In this case the formula matched by `phi -> psi` is consumed and the formula matched by `phi` is added to the antecedent and the formula matched by `psi` to the succedent. The sequent context we denote with Γ and Δ in text-book style rules is not mentioned explicitly in taclets.

assumes Like the *find* clause, the *assumes* clause of a taclet describes formulas that must be present in the sequent for the rule to be applied. The modus ponens rule

$$\mathsf{modusPonens}\ \frac{\Gamma,\ \phi,\ \psi \Rightarrow \Delta}{\Gamma,\ \phi,\ \phi \rightarrow \psi \Rightarrow \Delta}$$

could be expressed as a taclet using the assumes clause:

— Taclet ———

```
modusPonens{
  \assumes(phi ==> )
  \find(phi -> psi ==> )
  \replacewith(psi ==> )
}
```

——— Taclet —

Unlike the *find* clause, the formula matched by *assumes* is not consumed by the application of the taclet (unless this assumption also occurs in *find*).

add We can also define modusPonens in a “non-destructive” way such that its application does not consume the implication formula

$$\mathsf{modusPonensAdd}\ \frac{\Gamma,\ \phi,\ \psi,\ \phi \rightarrow \psi \Rightarrow \Delta}{\Gamma,\ \phi,\ \phi \rightarrow \psi \Rightarrow \Delta}$$

which is equivalent to the taclet

— Taclet ———

```
modusPonensAdd{
  \assumes(phi ==> )
  \find(phi -> psi ==> )
  \add(psi ==> )
}
```

——— Taclet —

It is also possible to combine one *add* clause with one *replacewith* clause.

Multiple Premises The rule for treating implications that are top level formulas in the antecedent possesses two premises:

$$\text{impRight}\ \frac{\Gamma \Rightarrow \phi,\ \Delta \qquad \Gamma,\ \psi \Rightarrow \Delta}{\Gamma,\ \phi \rightarrow \psi \Rightarrow \Delta}$$

In the taclet language multiple premises can be created by a semicolon separated list of multiple goal templates (each consisting of *add* and *replacewith* clauses):

— Taclet —

```
impRight{
  \find(phi -> psi ==> )
  \replacewith(==> phi);
  \replacewith(psi ==> )
}
```

— Taclet —

Rewrite Rules So far we have only considered rules applied to entire sequents. There are also rules that can be applied to terms irrespective of their location in the sequent, so-called rewrite rules. Let us consider the rewrite rule addZero

$$a + 0 \rightsquigarrow a$$

The *find* and *replacewith* parts of the corresponding taclet are not sequent templates like in the previous examples but term templates:

— Taclet —

```
addZero{
  \find(a+0)
  \replacewith(a)
}
```

— Taclet —

Rewrite rules can also be targeted to formulas and augmented with an assumes clause to define additional conditions necessary for their application.

closegoal Axioms such as the following

$$\text{trueRight}\ \frac{*}{\Gamma \Rightarrow \mathit{true},\ \Delta}$$

can be implemented using the *closegoal* construct:

— Taclet —

```
trueRight{
  \find(==> true)
  \closegoal
}
```

— Taclet —

sameUpdateLevel The flag *sameUpdateLevel* indicates that the entity matched by the *find* clause must occur in the same state as the formula matched by the *assumes* clause.

— Taclet —

```
applyEq{
  \assumes(s=t  ==>)\sameUpdateLevel
  \find(s)
  \replacewith(t)
}
```

— Taclet —

If `s` were occurring in different state than `s=t` we would not know if `s=t` still holds in this state.

Example 2.27 *The rule* applyEq *is not applicable to the sequent*

$$a = b \Rightarrow \langle \texttt{a++;} \rangle\, a = b$$

Since if the above sequent is evaluated in a state s, the first occurrence of $a = b$ is evaluated in s while the second is evaluated in $\rho(\texttt{a++;})(s)$.

Other Features In addition to what we have seen so far there are further mechanisms for defining conditions the instantiations of the used schema variables must fulfil or rule sets grouping the available rules in different categories that can be differently prioritised by the automatic proof finding strategy. We will not elaborate on these features here, a complete description of them can be found in the afore mentioned sources.

Taclets in Textbook-Style Notation

In this work a textbook-style-like notation for taclets is used. In the following it is defined how side conditions (*assumes* clauses) for rewrite rules and *find* or *assumes* clauses in non-rewrite rules matching non-top-level terms/formulas are expressed in this notation.

Non-Top-Level Terms and Formulas in Non-Rewrite Rules Let us consider the following taclet

— Taclet —

```
addAssumptionForR{
  \find(r)
  \add(P(r)  ==>)
}
```

— Taclet —

where `r` is a schema variable matching rigid terms and `P` is a predicate. The term matched by `r` can occur in arbitrary places in a sequent. To express this rule in a more textbook-style like notation we write:

$$\mathsf{addAssumptionForR}\ \frac{\Gamma, P(r) \Rightarrow \Delta}{\Gamma \Rightarrow \Delta} \ni r$$

Definition 2.28 (The relation $\ni$) *Let φ and ψ be* JAVA DL *formulas and t a term. We write*

$$\textsf{seqRule}\ \frac{\Gamma' \Rightarrow \Delta'}{\Gamma \Rightarrow \Delta} \ni \varphi$$

if the taclet seqRule *shall only be applied if either $\varphi \in \Gamma \cup \Delta$ or there is a formula Φ such that $\Phi \in \Gamma \cup \Delta$ and φ is subformula of Φ. Analogously we define that*

$$\textsf{seqRule}\ \frac{\Gamma' \Rightarrow \Delta'}{\Gamma \Rightarrow \Delta} \ni t$$

shall only be applied if there is a formula Φ such that $\Phi \in \Gamma \cup \Delta$ and t is subterm of Φ. If multiple formulas (or terms) are required to occur in the conclusion we write:

$$\textsf{seqRule}\ \frac{\Gamma' \Rightarrow \Delta'}{\Gamma \Rightarrow \Delta} \ni \varphi, \psi$$

Note, that $\ni r$ used in rule addAssumptionForR merely constrains the situation in which this rule should be applied (important for proof automation), namely if r occurs in the regarded sequent, but do *not* express that the fact that r occurs in the conclusion is a logical consequence of the premise(s).

A flexible term or formula specified by $\ni$ to occur in the conclusion of the rule must always occur in the same state as the formulas added to the premises.

Rewrite Rules A rewrite taclet in its basic form, without additional assumption necessary for its application, is denoted as follows (exemplified by the already known taclet addZero):

$$\textsf{addZero} \quad a + 0 \rightsquigarrow a$$

Side conditions that must be satisfied when applying a rewrite rule as it is, for instance, the case for the rule applyEq shown in the previous Section are expressed as follows:

$$\textsf{applyEq}\ \frac{[s \rightsquigarrow t]}{\Gamma,\ s = t \Rightarrow \Delta}$$

The semantics of the above rule is, that in a sequent $\Gamma,\ s = t \Rightarrow \Delta$ arbitrary occurrences of s occurring on the same update level as $s = t$ can be replaced by t. If an additional side condition $\ni r$ (or $\ni \phi$) is specified, r (ϕ) must also occur in the same state as the target of the rewrite rule.

A rewrite rule is correct if the validity of each sequent obtained by applying the rewrite rule entails the validity of the original sequent.

2.3.2 Symbolic Execution

For reasoning about programs we need to compute the state change caused by a program. In the JAVA DL calculus this is done by symbolic execution [King, 1976]. Symbolic execution denotes a technique for executing a program with, as the name suggests, symbolic instead of concrete values. The semantics of JAVA required for symbolic execution is encoded in taclets. We can basically distinguish two kinds of symbolic execution taclets:

- Taclets flattening statements (i.e., decomposing complex statements into simpler ones). The order in which a complex statement is decomposed matches the evaluation order specified by the *Java Language Specification*
- Taclets compiling not further decomposable statements to updates and case distinctions.

Both kinds of taclets operate solely on the first active statement of a program (i.e., the first statement occurring after a non-active prefix of opening braces, method-frames (see Section 2.3.6) and **`try`** blocks). In rule schemata we denote the inactive prefix with π and the remainder of the program succeeding the first active statement with ω.

Example 2.29 (Symbolic Execution Taclets) *The taclet* assignmentUnfoldRight *flattens an attribute access nse.a on the right-hand-side of an assignment, where nse is a non-simple expression (an expression potentially having side effects):*

$$\mathsf{assignmentUnfoldRight}\ \frac{\Gamma \Rightarrow \{\mathcal{U}\}\langle\pi\ \ T_{nse}\, v_0 = nse;\ v = v_0.a;\ \omega\rangle\, \phi,\ \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}\langle\pi\ \ v = nse.a;\ \omega\rangle\, \phi,\ \Delta}$$

The taclets assignmentReadAttribute *computes the effect of assignments of the form* $v_0 = v.a;$ *where* v *and* v_0 *are variables and* a *is an attribute. The assignment* $v_0 = v.a;$ *cannot be further decomposed and its effect is thus described by an update necessitating a case distinction on whether* $v \doteq$ ***`null`*** *holds:*

$$\mathsf{assignmentReadAttribute}\ \frac{\begin{array}{l}\Gamma \Rightarrow \{\mathcal{U}\}v \doteq \mathtt{\textbf{null}},\ \{\mathcal{U}\}\{v_0 := v.a\}\langle\pi\ \ \omega\rangle\, \phi,\ \Delta\\ \Gamma,\ \{\mathcal{U}\}v \doteq \mathtt{\textbf{null}} \Rightarrow \{\mathcal{U}\}\langle\pi\ \ throw\ new\ NullPointerException();\ \omega\rangle\, \phi,\ \Delta\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle\pi\ \ v_0 = v.a;\ \omega\rangle\, \phi,\ \Delta}$$

2.3.3 Object Creation

Having Java as the target language we need to model object creation and initialisation. As many other dynamic logics, and modal logics in general, Java DL operates under the technically advantageous constant-domain assumption. This means that all states of a Kripke structure share the same universe, which seems contradictory to modeling dynamic object creation since it is not possible to add new elements to the universe.

Remark 2.30 (Benefits of the Constant-Domain Assumption) *Due to the* constant-domain assumption*, the range of quantifiers is not state-dependent. Thus the formulas*

$$\forall x; [p]\phi \leftrightarrow [p]\forall x; \phi$$

and

$$\forall x; \{\mathcal{U}\}\phi \leftrightarrow \{\mathcal{U}\}\forall x; \phi$$

hold. This allows us to propagate updates into the scope of quantifiers which is essential for the application of updates.

The implications this carries for the modeling of object creation are that all objects that can ever be created by a program have to exist in *every* program state. Whether an object is created is determined by the values of certain *implicit* (see Remark 2.33) fields, when a new object is created these fields have to be changed appropriately. To formalize this approach we first define the notion of object repositories and repository access functions.

Definition 2.31 (Object Repository) *Let C be a non-abstract class type. The object repository Rep_C denotes the set of all elements of* $\mathbf{U}$ *of dynamic type C:*

$$Rep_C := \{e \in \mathbf{U} \mid \delta(e) = C\}$$

where Rep_C is an infinite and enumerable set.

Since we defined object repositories to be enumerable sets (which is actually a restriction we imposed on the set of admissible states in Java DL Kripke structures), it is possible to index them and provide by this a means to access every repository object including the not yet created ones. This indexing is done by a *repository access function.*

Definition 2.32 (Object Repository Access Function) *Let $(\mathcal{M}, \mathcal{S}, *, \rho)$ be a* Java DL *Kripke structure with $\mathcal{M} := (\mathcal{D}, I)$ and C a non-abstract class type. Then the* object repository access function *get_C is a rigid unary function symbol with*

$$I(get_C) \;:\; \mathbb{Z} \to Rep_C$$

being a surjective mapping and $I(get_C)|_{\mathbb{N}_0}$ being bijective.

Now as we have a means to talk about all (created and non-created) object we need to able to distinguish which objects are already created and which are still "available" when a new instance is to be created. For this we introduce the static field <`nextToCreate`> (which we abbreviate <`ntc`> in the following) for each non-abstract class type C denoting the smallest non-negative index such that the object $get_C(C.$<`ntc`>$)$ is not created. In addition we require that all objects of dynamic type C having an index greater $C.$<`ntc`> are not created either. Thus in each state s the set of created objects is

$$\{s(get_C)(i)|\; i \in \mathbb{Z} \wedge 0 \leq i < val_s(C.\texttt{<ntc>})\}$$

For convenience reasons we define in addition a boolean instance field <`created`> (which we abbreviate <`c`> in the following) in `java.lang.Object` which evaluates to *TRUE* if (and only if) the corresponding object is created.

An instance creation expression is symbolically executed by replacing it by a sequence of calls to implicit methods modeling object creation and initialisation which we do not consider in-depth here (a detailed description of the handling of object creation in Java DL can be found in Chapter 3 of [Beckert et al., 2007]). The object allocation itself is represented by the invocation of the implicit method `C.<allocate>` which returns the object $get_C(C.$<`ntc`>$)$ increases the $C.$<`ntc`> pointer and sets the <`c`> attribute of the returned object to *TRUE*. This behavior is encoded in the taclet allocate:

$$\text{allocate}\ \frac{\begin{array}{l}\Gamma \Rightarrow \{\mathcal{U}\}\{v := get_T(T.\texttt{<ntc>}) \;||\\ \qquad\qquad T.\texttt{<ntc>} := T.\texttt{<ntc>} + 1 \;||\\ \qquad\qquad get_T(T.\texttt{<ntc>}).\texttt{<c>} := \mathit{TRUE}\}\langle \pi\, \omega\rangle\, \phi,\ \Delta\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle \pi\ \texttt{v=T.<allocate>();}\ \omega\rangle\, \phi,\ \Delta}$$

Remark 2.33 (Implicit Fields and Methods) *In* Java DL *the terms* implicit field *and* implicit method *refers to fields and methods which do not exist in the original code and are added for providing additional state information or describing certain operations during object creation. Implicit fields/methods are enclosed in pointed brackets for distinguishing them from "non-implicit" fields/methods.*

Object creation and initialisation is, for instance, modeled by the four implicit methods

- ***private static*** *`<allocate>()`: allocating a fresh object from the object repository,*
- ***protected void*** *`<prepare>()`: assigning default values to all instance fields (such as **null** for reference type instance fields),*
- *mods **void** `<init>(params)`: executing the initialisers of instance fields and the body of the invoked constructor and*
- ***public static*** *T `<createObject>()`: setting the values of certain implicit fields related to object initialisation and invoking the methods `<allocate>` and `<prepare>`.*

2.3.4 Reachable States

Tightly coupled with our modeling of object creation is the issue of states reachable by a Java program. Properties that hold in each state reachable by a Java program are for instance that

- only a finite number of objects is created
- objects referenced by instance attributes of created objects are either **`null`** or created.

We cannot simply restrict the states contained in a Java DL Kripke structure only to those states reachable by a Java programs since it is possible to build updates resulting in non-reachable states. Given, for instance, an arbitrary state s and the update

$$\{\mathcal{U}\} := \{\text{for java.lang.Object } o;\ \text{if}(\mathit{true})\ o.\texttt{<c>} := \mathit{TRUE}\}$$

the state $val_s(\mathcal{U})(s)$ is not reachable by any Java program since the number of objects created in $val_s(\mathcal{U})(s)$ is infinite.

The properties characterizing a reachable state can be axiomatized as a set of Java DL formulas (see Chapter 3 of [Beckert et al., 2007]). Thus we could add this set of axioms each time we need to specify that a certain state is reachable (such as in the preconditions of method contracts). This would, however, lead to impracticably lengthy formulas. Instead we introduce a heap-dependent predicate RS (short for *inReachableState[*]*) that holds in exactly those states reachable by a Java program. The properties holding in these states are axiomatized by taclets that can be applied in states in which RS is known to be true. The following taclet, for instance, states that each object referenced by an instance attribute of a created object is either **`null`** or created:

$$\mathsf{instAttrNullOrCreated}\ \frac{\Gamma,\ o.\texttt{<c>} \doteq \mathit{TRUE},\ \mathrm{RS},\ o.a.\texttt{<c>} \doteq \mathit{TRUE} \vee o.a \doteq \texttt{null} \Rightarrow \Delta}{\Gamma,\ o.\texttt{<c>} \doteq \mathit{TRUE},\ \mathrm{RS} \Rightarrow \Delta}\ \ni o.a$$

where $o.a$ is a reference type attribute term.

In certain cases it is necessary to show that RS is satisfied in a specific state: When, for instance, a method contract is applied (see Section 2.3.5), we need show that its precondition, which usually incorporates the RS predicate, holds in the pre-state of the method invocation. In this situation we have to show that a sequent of the form

$$\Gamma,\ \mathrm{RS} \Rightarrow \{\mathcal{U}\}\,\mathrm{RS},\ \Delta$$

is valid. This could be solved by replacing RS by a conjunction of the already mentioned axioms describing reachable states. This is again impracticable due to the size of this formula. Instead the update $\{\mathcal{U}\}$ is analyzed and RS is replaced only by those formulas whose truth value is potentially affected by $\{\mathcal{U}\}$.

2.3.5 Method Contracts and Class Invariants

The behavior of a method (or a constructor) is described by a set of method contracts [Meyer, 1992]. In the scope of the KeY system method contracts serve different purposes:

- The correctness of the implementation of a method with respect to its contract can be verified by KeY. For this, different kinds of proof obligations (POs) exist encoding different aspects of the contract. For instance, there is a proof obligation for showing that the method establishes its postcondition whenever called in a state satisfying the corresponding precondition.
- Contracts can be used to approximate the effect of a method call which makes symbolic execution and thus also verification of programs modular. The soundness of a proof constructed by doing so, of course, depends on the correctness of the used contracts.

In this work we only consider total correctness method contracts (i.e., method contracts requiring termination of the specified method). Partial correctness is handled analogously but slightly simpler.

Definition 2.34 (Method Contracts) *A method contract for an operation (i.e., a method or constructor) op is quintuple*

$$(Pre, Post, Mod, Dep, Cap)$$

where

- *$Pre \in Fml$ denotes the precondition of op and may contain the following program variables:*
  - *in case op is an instance method or a constructor the program variable self denotes its receiver object, i.e., the object the method is invoked on, or (in case op is a constructor) the object initialised by the constructor.*
  - *$p_1, \ldots, p_n$ represent the parameters of op.*

  Pre may not contain free logic variables.
- *$Post \in Fml$ is the postcondition of op that has the form:*

$$(exc \doteq \mathbf{null} \rightarrow \phi) \wedge (exc \not\doteq \mathbf{null} \rightarrow \psi)$$

  where ϕ is the postcondition for the case op terminates normally and ψ for the case of an abrupt termination (by an exception). The program variable exc denotes the exception possibly thrown by op. If no exception is raised by op $exc \doteq \mathbf{null}$ holds. The program variable exc may occur in the exceptional postcondition ψ but not in ϕ. In turn the program variable result (denoting the return value of op in case op has one) may only occur in the normal termination postcondition ϕ. Besides exc and result, Post may also contain the program variables self (for non-static methods) and $p_1, \ldots, p_n$. As Pre, Post may not contain free logic variables.
- *Mod is a set of location descriptors specifying the modifies (or also: assignable) clause of op (i.e., all locations that might be changed by op when called in a state satisfying Pre).*

- *Dep, which is also a set of location descriptors, specifies the depends clause for op under the precondition pre (i.e., all locations whose value can influence the returned result and side effects of op when called in a state satisfying Pre).*

- *finally the set of location descriptors Cap specifies all reference type locations whose value can be captured by the execution of op (when started in a state satisfying Pre). We say that an object o is captured by op if in the post-state o is referenced by a location it was not referenced by in the pre-state.*

Again, self and $p_1, \ldots, p_n$ *can occur in* Mod*, Dep and* Cap*.*

The semantics of method contracts permits to underspecify (or, in other words: over-approximate) Mod, *Dep* and Cap clauses. This means that the union of a correct Mod (*Dep*, Cap) clause with an arbitrary set of location descriptors is still a correct Mod (*Dep*, Cap) clause.

In case the *depends* and *captures* clause are not relevant for certain considerations we may omit them and specify a contract as the triple $(Pre, Post, Mod)$. For only partly specified contracts (see also Section 3) the defaults for the unspecified clauses are chosen as the most permissive values for them which is *true* for Pre and $Post$ and the entire heap $*$ for Mod, *Dep* and Cap.

Example 2.35 *Let us consider the method:*

— Java —

```java
public void swap(MyObject a, MyObject b){
  Object tmp = a.attr;
  a.attr = b.attr;
  b.attr = tmp;
}
```

— Java —

A possible method contract for this method is given by:

$$
\begin{array}{l}
(a \not\doteq \mathbf{\mathtt{null}} \land b \not\doteq \mathbf{\mathtt{null}},\\
\;(exc \doteq \mathbf{\mathtt{null}} \rightarrow a.attr \doteq b.attr^{@pre} \land b.attr \doteq a.attr^{@pre}) \land (exc \not\doteq \mathbf{\mathtt{null}} \rightarrow false),\\
\;\{a.attr, b.attr\},\\
\;\{a.attr, b.attr\},\\
\;\{a.attr, b.attr\})
\end{array}
$$

Since swap does not throw an exception under the chosen precondition, our exceptional postcondition is just given by the formula $false$ *and thus* $exc \not\doteq \mathbf{\mathtt{null}} \rightarrow false$ *logically equivalent to* $exc \doteq \mathbf{\mathtt{null}}$*. Therefore, the entire postcondition is equivalent to*

$$a.attr \doteq b.attr^{@pre} \land b.attr \doteq a.attr^{@pre}$$

Another important means beside method contracts for specifying object oriented programs are class invariants. A class invariant is a property that must be satisfied by a class or all instances of a class in certain states[6].

[6] Which states these are in detail is exemplarily outlined in the following. For a comprehensive elaboration on this topic refer to [Roth, 2006].

Definition 2.36 *An invariant $Inv \in Fml$ is satisfied by a class C if each operation op of C executed in a state satisfying at least the precondition of one of op's contracts preserves Inv.*

Example 2.37 *Let us consider the class*

— *Java* —

```java
public class Wrapper{
  Object obj;

  public Wrapper(Object o){
    obj = o;
  }

  public Object getObject(){
    return obj;
  }
}
```

— *Java* —

where

$$(o \not\doteq \mathtt{null}, true, \{self.obj\}, \{o\}, \{o\})$$

is the (only) contract for the constructor `Wrapper(Object o)` *and*

$$(true, result \doteq self.obj, \emptyset, \{self.obj\}, \emptyset)$$

the contract for the method `getObject()`*. Then the class $Wrapper$ satisfies the invariant:*

$$\forall\, Wrapper\ w;\ (w.\text{<c>} \doteq TRUE \rightarrow w.obj \not\doteq \mathtt{null})$$

The Method Contract Rule

As already hinted, a method contract can add to modularize symbolic execution by approximating a method call by one of the called method's contracts. This approach can of course only be sound if the used method contract is correct (an issue that is addressed in Section *2.3.5-Proof Obligations*). The rule for applying the contract for an instance method to its invocation inside a diamond modality looks as follows:

methodContractInst

$$\frac{\begin{array}{l}\Gamma \Rightarrow \{\mathcal{U}\}\{self := se_{rec} || p_1 := a_1 || \ldots || p_n := a_n\}Pre',\ \Delta \\ \Gamma,\ \{*_i^{Mod}\}exc \doteq \mathtt{null} \Rightarrow \\ \quad \{\mathcal{U}\}\{*_i^{Mod} ||\ self := se_{rec} || p_1 := a_1 || \ldots || p_n := a_n\}(Post' \rightarrow \{lhs := result\}\langle \pi\ \ \omega\rangle\, \phi),\ \Delta \\ \Gamma,\ \{*_i^{Mod}\}exc \not\doteq \mathtt{null} \Rightarrow \\ \quad \{\mathcal{U}\}\{*_i^{Mod} ||\ self := se_{rec} || p_1 := a_1 || \ldots || p_n := a_n\}(Post' \rightarrow \langle \pi\ \ throw\ exc;\ \omega\rangle\, \phi),\ \Delta\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle \pi\ \ lhs = se_{rec}.m(a_1, ..., a_n)@(C);\ \omega\rangle\, \phi,\ \Delta}$$

where $(Pre, Post, Mod)$ is a contract for the method invoked by[7] $se_{rec}.m(a_1, ..., a_n)@(C)$; and $\{*_i^{Mod}\}$ is an anonymising update for Mod with $*_i$ as well as $*_i^{Locs}$ (for any set of location

[7] The statement $lhs = se_{rec}.m(a_1, ..., a_n)@(C)$; is not a "normal" method invocation but a so-called *method body statement* which is basically a placeholder for the body of the invoked method m which is implemented in class C. Section 2.3.6 elaborates on the semantics and purpose of *method body statements*.

descriptors *Locs*) not yet occurring in the sequent the rule is applied to. This anonymising update is used to approximate the state change caused by m by setting all locations it can possibly modify to unknown values. We define

$$\begin{aligned} Pre' &:= Pre \land Conj_{Assumed} \\ Post' &:= Post \land Conj_{Ensured} \end{aligned}$$

where $Conj_{Assumed}$ is a conjunction of invariants we assume to hold in the pre-state of m's method invocation and $Conj_{Ensured}$ is a conjunction of invariants being established by m in the respective post-state. Section *2.3.5-Proof Obligations* describes and motivates the criteria relevant for the choice of $Conj_{Assumed}$ and $Conj_{Ensured}$.

The first premise states that the precondition Pre' has to hold in the state the method is invoked in. In this state the receiver object and the arguments of the method are given by se_{rec} and $a_1, \ldots, a_n$ so we update the placeholders *self* and $p_1, \ldots, p_n$ occurring in Pre with these values.

The second premise represents the normal termination case: We assume that no exception (that is not internally caught by m) is raised by m and thus $\{*_i^{Mod}\}exc \doteq \texttt{null}$ holds. In this case we have to show that the postcondition $Post'$ implies that after the execution of the remaining program ϕ holds.

In the third premise we assume that the method terminated by an exception and thus $exc \not\doteq \texttt{null}$ and $Post'$ holds. In this case after the execution of π *throw exc*; ω the formula ϕ has to hold.

Corresponding rules for the invocation of static methods or methods without return value are defined analogously.

Proof Obligations

The KeY system offers a wide range of proof obligations (POs) for verifying the compliance of a program with its specification (so-called vertical proof obligations). In the following we provide short a summary of the proof obligations relevant for this work. A detailed description of vertical as well as horizontal proof obligations can be found in [Roth, 2006] and [Beckert et al., 2007].

First we need to define some basic building blocks required for all of the following POs. If we recall the structure of postconditions (see previous section) the need for capturing exceptions thrown by the regarded program becomes obvious (otherwise we could not "talk" about them in the postcondition). To achieve this we wrap the specified method or constructor *op* in a *try-catch* statement:

— Java —

```java
exc=null;
try{
  Prg(op;self;p1,...,pn;result);
}catch(Throwable e){
  exc = e;
}
```

— Java —

Method/Constructor `op` declared in `T`	`Prg(op;self;p1,...,pn;result)`
`T0 m(T1,...,Tn)`	`result=self.m(p1,...,pn)@(T);`
void `m(T1,...,Tn)`	`self.m(p1,...,pn)@(T);`
static `T0 m(T1,...,Tn)`	`result=T.m(p1,...,pn)@(T);`
static void `m(T1,...,Tn)`	`T.m(p1,...,pn)@(T);`
`T(T1,...,Tn)`	`result =` **new** `T(p1,...,pn)@(T);`

Table 2.1: Programs in proof obligations

where `Prg(op;self;p1,...,pn;result)` is a placeholder for the statement defined by table 2.1. In the following we abbreviate the above program with $Prg_{op}()$.

In Section 2.3.4 the RS predicate is defined that characterizes the states reachable by a program. For defining the states in which a method call can occur we have to constrain this a bit further which is done by the formula $ValidCall^{sem}_{op}$ (with $op := self.m(p_1, \ldots, p_n);$) characterizing the legal pre-states for the method invocation op under the semantics sem determined by the considered programming language. This work distinguishes in this context the semantics $Java$ and SCJ (for *Safety Critical Java*). For the $Java$ semantics $ValidCall^{Java}_{op}$ is given by:

$$ValidCall^{Java}_{op} := \begin{array}{c} \mathrm{RS} \wedge self \not\doteq \texttt{null} \wedge self.\texttt{<c>} \doteq TRUE \wedge \\ \bigwedge\limits_{\substack{i:1\leq i\leq n \text{ and} \\ \alpha(p_i) \text{ is ref. type}}} (p_i \doteq \texttt{null} \vee p_i.\texttt{<c>} \doteq TRUE) \end{array}$$

Now we can start to define POs for the various properties a program has to fulfil. The PO $PreservesInv(op; Assumed; Ensured)$ states that op preserves the set of invariants $Ensured$ if executed in a (legal) state satisfying at least the precondition of one contract of op and the set of invariants $Assumed$ (typically $Ensured \subseteq Assumed$ holds):

$$\begin{array}{l} PreservesInv(op; Assumed; Ensured) := \\ \quad Conj_{Assumed} \wedge Disj_{Pre} \wedge ValidCall^{sem}_{op} \rightarrow [Prg_{op}()]\, Conj_{Ensured} \end{array}$$

where $Disj_{Pre}$ is the disjunction of all preconditions for op and $Conj_F$ is the conjunction of all formulae in set F.

The PO $EnsuresPost(ct; Assumed)$ states that the postcondition $Post$ (as defined by ct) is ensured by the corresponding operation op under assumption of the invariants $Assumed$:

$$EnsuresPost(ct; Assumed) := Conj_{Assumed} \wedge Pre \wedge ValidCall^{sem}_{op} \rightarrow \langle Prg_{op}() \rangle\, Post$$

In the previous section we defined the rule methodContractInst which uses a method contract whose pre- and postcondition was strengthened by the invariants $Assumed$ resp. $Ensured$. As already stated, the correctness of the applied method contract is vital for the correctness of the application of rule methodContractInst. The PO $EnsuresPost(ct; Assumed; Ensured)$ can be used to check that op ensures the postcondition $Post$ and the invariants $Ensured$ when executed in a state satisfying Pre and $Assumed$:

$$\begin{array}{l} EnsuresPost(ct; Assumed; Ensured) := \\ \quad Conj_{Assumed} \wedge Pre \wedge ValidCall^{sem}_{op} \rightarrow \langle Prg_{op}() \rangle\, Post \wedge Conj_{Ensured} \end{array}$$

In addition we also have to verify that the *assignable* clause Mod is correct. Section 9.2.1 elaborates on the verification of *assignable*, *depends* and *captures* clauses.

Having defined $EnsuresPost(ct; Assumed; Ensured)$ it becomes clear how $Conj_{Assumed}$ and $Conj_{Ensured}$ should be chosen when applying a method contract: $Conj_{Assumed}$ has to be chosen strong enough to make the implications (occurring in the second and third premise of rule methodContractInst)

$$Post' \quad \rightarrow \quad \langle \pi \;\; throw\, exc;\; \omega \rangle\, \phi \tag{2.18}$$

$$Post' \quad \rightarrow \quad \langle \pi \quad \omega \rangle\, \phi \tag{2.19}$$

true in the states they are evaluated in (which are constraint by the sequent contexts). $Conj_{Assumed}$ on the other hand must be chosen strong enough to entail validity of formula $EnsuresPost(ct; Assumed; Ensured)$ otherwise the application of methodContractInst is unsound.

Further strengthening the formula $Conj_{Ensured}$ could make it easier to prove validity of the sequents resulting from the second and third premise of rule methodContractInst (since we have stronger assumptions then). In turn, however, this might necessitate to strengthen also $Conj_{Assumed}$ for preserving validity of $EnsuresPost(ct; Assumed; Ensured)$. By doing so we would also strengthen the formula Pre' occurring in the first premise of methodContractInst which could render the corresponding sequent invalid or at least less easier to prove valid.

2.3.6 Inlining of Method Bodies

In Section 2.3.5 a possibility for symbolically executing a method invocation is discussed: using a method contract for approximating the state change caused by the method invocation. This section reviews an alternative handling of method calls namely inlining the method body. The disadvantage this carries is the obvious lack of modularity (at least for dynamically bound methods): A proof performed using method inlining is only valid for a closed program and becomes invalid whenever new implementations of an inlined method are added to the regarded program.

We now consider the symbolic execution of a non-*void* instance method invocation. As described in Section 2.3.2, statements containing complex subexpressions are first flattened. For a method invocation this results in a statement of the form (we ignore additional statements introduced by the process of flattening here):

— Java —

```
 T v = self.m(v1,...,vn);
```

— Java —

where `v`, `self`, `v1` ... `vn` are program variables.

The next step is performing a case distinction (encoded as an **`if`**-cascade) on the dynamic type (only subtypes of α(`self`) implementing `m` have to be considered here) of the receiver object `self`:

— Java —

```
if(self instanceof T1){
    v=self.m(v1,...,vn)@(T1);
}else if(self instanceof T2) {
    v=self.m(v1,...,vn)@(T2);
}else if ...
...
}else{
    v=self.m(v1,...,vn)@(Tm);
}
```

JAVA

where the statements `v=self.m(v1,...,vn)@(Ti)` are so-called *method-body-statements* which serve as placeholders for a concrete method body.

Definition 2.38 (Method Body Statement) *Let* `r`*,* `v1`*,...,*`vn` *and* `self` *be program variables. Then a* method body statement *is a statement of the form:*

```
r=self.m(v1,...,vn)@(T);
```

with class `T` *implementing a non-void instance method with signature* $\texttt{m}(\alpha(\texttt{v1}), \ldots, \alpha(\texttt{vn}))$ *or:*

```
r=m(v1,...,vn)@(T);
```

with class `T` *implementing a non-void static method with signature* $\texttt{m}(\alpha(\texttt{v1}), \ldots, \alpha(\texttt{vn}))$ *or:*

```
self.m(v1,...,vn)@(T);
```

with class `T` *implementing a void instance method with signature* $\texttt{m}(\alpha(\texttt{v1}), \ldots, \alpha(\texttt{vn}))$ *or:*

```
m(v1,...,vn)@(T);
```

with class `T` *implementing a void static method with signature* $\texttt{m}(\alpha(\texttt{v1}), \ldots, \alpha(\texttt{vn}))$*.*

A method body statement is symbolically executed by replacing it with the method body it stands for and enclosing it in a *method frame* (a statement keeping track of the current execution context).

Definition 2.39 (Method Frame Statement) *Let* `r` *and* `self` *be program variables,* `T` *a class type and* `p` *a sequence of statements then*

```
method-frame(result->r,
             source=T,
             this=self) : { p }
```

is a method frame *statement.*

Replacing a method-body statement with the actual method body it stands for is done by the rule methodBodyExpand:

$$\text{methodBodyExpand}\ \frac{\Gamma \Rightarrow \{\mathcal{U}\}\langle \pi\ \texttt{method-frame(result->res,}\ \texttt{source=T,}\ \texttt{\textbf{this}=self)} : \{\texttt{body}\}\ \omega\rangle\phi,\ \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}\langle \pi\ \texttt{res=self.m(v1,...,vn)@(T);}\ \omega\rangle\ \phi,\ \Delta}$$

where *body* is a sequence of assignments assigning the variables `v1`, ..., `vn` to the formal parameters used in the corresponding implementation of `m` followed by the method body itself.

2.3.7 Inner Classes

Typical RTSJ applications make extensive use of inner classes which necessitated support for this Java construct in Java DL. This section provides a quick overview of the different kinds of inner classes [Gosling et al., 2005, Igarashi and Pierce, 2002] existing in Java and how they are handled in Java DL.

Member, Local and Anonymous Classes in Java

Java distinguishes 4 different kinds of inner classes:

- Static inner classes (also known as top-level inner classes)
- Non-static inner classes
- Local classes
- Anonymous (local) classes

Static Inner Classes A static inner class is declared as a static member of a class. It has access to all static (but no instance) fields and methods of the enclosing class.

— Java

```
public class A{
  ...
  private static int sa;
  ...
  public static class B{
    void m(){int x = sa; ...}
  }
}
```

Java —

A static class can be instantiated in the same manner as an ordinary top level class using its full name which is the class name prefixed with the names of the enclosing classes. A new instance of class `B`, for instance, shown in the above example is created by `new A.B()`.

Non-Static Inner Classes Non-static inner classes (we just refer to them as inner classes in the following) are declared as instance members of an enclosing class. Like static inner classes, they can access static fields and methods of the enclosing class. In addition, each instance of an inner class is associated with an instance of the enclosing class and can access all instance fields and methods of this instance.

— Java

```
public class A{
  private int ia;
  private static int sa;

  class InnerA{
    int m(){return sa+ia;}
  }
}
```

JAVA

For resolving field references that are not explicitly prefixed the field is looked up first in the most inner instance and then successively in the outer instances iterating from inner to outer instances. In the above example, the field `ia` accessed in method `m()@(InnerA)` is first looked up in `InnerA` and since not being found there in `A` afterward. So we can think of this occurrence of `ia` as prefixed by an implicit **`this`** reference to the enclosing **`this`** object. It is also possible to reference the enclosing instance explicitly by `A.this` (thus in the above example we could also write `A.this.ia`). In case the nesting level is greater 1 like here:

```
public class A{... class B{... class C{...} ...} ...}
```

every transitively reachable enclosing instance can be accessed analogously. In the context of class `C` we can for instance write `A.this` to access the instance of `A` that encloses the instance of `B` which is the enclosing instance of the regarded `C` object (and thus accessible through `B.this`).

When creating an instance of an inner class, an appropriate enclosing instance must be provided. This can either happen

- implicitly by creating the instance of the inner class in a non-static context of the enclosing class (in this case the enclosing instance is the object referenced by **`this`** when the inner class is instantiated).
- or explicitly by providing an object `a` of the outer class and using a special variant of the new operator: `a.new InnerA()`.

Both alternatives are shown in the following example:

JAVA

```
public class B{
  private int ia;

  public int m(){
    A a = new A();
    A.InnerA ina = a.new InnerA();
    InnerB ib = new InnerB(a);
    return ib.m();
  }

  class InnerB extends A.InnerA{
    public InnerB(A a){ a.super();}
  }
}
```

JAVA

When an instance of the inner class `InnerB` is created in the above example, its enclosing instance of type `B` is set to the object denoted by **`this`**. Since `InnerB` extends another inner class, `InnerA`, and requires as such also an enclosing instance of type `A`, we need to provide one which is done by the **`super`** constructor call `a.super()`. Thus the newly created object `ib` possesses two enclosing instances (one of type `A` and one of type `B`). In Java, field references as well as implicit references to enclosing instances are bound statically. This means for the method call `ib.m()` in the above example that it returns the value of[8] `A.sa@(A)+A.this.ia@(A)` since it is implemented in `A.InnerA`.

Local Classes Unlike inner classes local classes are not declared as members of a class but inside a Java block (similar to a local variable). They are local in the sense that they are only visible in the block they are declared in (in the code following their declaration). Like inner classes they possess an enclosing instance but can in addition access *final* local variables visible at the point they are declared.

— Java —

```java
public class A{
  int i = 1;

  public int m(){
    final int il = 0;
    class LocalInM{
      public int m(){ return i+il;}
    }
    LocalInM l = new LocalInM();
    return l.m();
  }
}
```

— Java —

Anonymous Classes An anonymous class is in principle a local class without a name. In the following example an instance of an anonymous class extending an existing class `B` is created:

— Java —

```java
public class A{
    int i = 1;

    public int m(){
        final int il = 0;
        B b = new B(){
            public int m(){ return i+il;}
        };
        return b.m();
    }
}
```

— Java —

The object created by the constructor call `new B() ...` in the above code has an anonymous dynamic type that is a direct subtype of `B`.

[8] The notation `a@(T)` again refers to the attribute `a` declared in class `T`.

Handling in Java DL

Static inner classes can be treated like top-level classes. The access to static members of the enclosing classes poses no problem since static binding is used for this. For the handling of non-static inner classes in Java DL two issues are of special importance:

- determining the correct enclosing instance implicitly or explicitly referenced in an inner class and
- resolving references to final local variables in local (and anonymous) classes.

The Implicit Field `<enclosingThis>` For storing the enclosing instance of an inner class we define an implicit field `<enclosingThis>`. This field is initialised by the implicit method `<init>` taking the enclosing instance as an argument which needs to be taken into account when evaluating a constructor call:

$$\mathsf{instanceCreationInner}\ \frac{\Gamma \Rightarrow \langle \pi\ \begin{array}{l}\texttt{T v0 = T.<createObject>();}\\ \texttt{v0.<init>(args, o)}\\ \texttt{v0.<initialized> = }\textbf{true}\texttt{;}\\ \texttt{v = v0;}\\ \omega\ \rangle \phi,\ \Delta\end{array}}{\Gamma \Rightarrow \langle \pi\, \texttt{v = o.}\textbf{new}\texttt{ T(args);}\, \omega\rangle\ \phi,\ \Delta}$$

For implicitly provided enclosing instances which can be considered syntactic sugar for an enclosing instance explicitly provided by

```
this.new ...
```

we proceed analogously.

When symbolically executing a program, the `<enclosingThis>` pointer is used to resolve implicit and explicit *this* pointers to enclosing classes which is needed for resolving field and method references[9]. Let `EnclosingT.this` be a **`this`** pointer `InnerC` the static context `EnclosingT.this` is evaluated in and `self` the object referenced by **`this`**. The resolution of an explicit *EnclosingT.this* pointer is described by the algorithm (in pseudo code):

— Pseudo Code —

```
result = self;
staticContext = InnerC;
while(staticContext != EnclosingT){
  result = result.<enclosingThis>@(staticContext);
  staticContext = staticTypeOf(result);
}
```

— Pseudo Code —

For resolving implicit enclosing **`this`** pointers we have to change the above algorithm slightly since only the class in which the field/method implicitly prefixed by this pointer is declared in was determined at parse-time. The *static* type of the corresponding enclosing instance can be a subtype of it:

[9] We assume that static binding and also the determination of the type of an implicit **`this`** pointer is already performed at parse-time and the this information is available to the calculus. Thus this information does not need to be computed during its symbolic evaluation.

— Pseudo Code —

```
result = self;
staticContext = InnerC;
while(!staticContext subtypeof EnclosingT){
  result = result.<enclosingThis>@(staticContext);
  staticContext = staticTypeOf(result);
}
```

— Pseudo Code —

Example 2.40 (Resolving of Enclosing *this* pointers) *Let* `A` *be a top-level class containing the inner class* `InnerA` *and declaring the field* `i`*:*

— Java —

```
public class A{
  int i=1;

  class InnerA{
    void m(){
      int c = i;
    }
  }
}
```

— Java —

where `InnerA` *does not declare a field named* `i`*. Then the symbolically executing the assignment*

$$\langle \texttt{\{method-frame(source=A.InnerA, } \textbf{this}\texttt{=self): \{ c = i;\}\}}\rangle\phi$$

results in the following update:

$$\{c := self.{<}enclosingThis{>}@(A.InnerA).i\}$$
$$\langle \texttt{\{method-frame(source=A.InnerA, } \textbf{this}\texttt{=self): \{\}\}}\rangle\phi$$

References to Final Local Variables Local and anonymous classes can reference final local variables of the enclosing block. We handle this by a program transformation:

- For each final local variable accessed by the local class we add a new uniquely named instance field.
- The constructors of the local classes are extended to initialise these new attributes
- The signature of the constructors is changed accordingly.

Example 2.41 (Handling of Final Local Variables) *Let us consider the following method incorporating a local class:*

— Java —

```
public int m(){
  final int i1 = 0;
  class B{
    public int m(){
      return i+il;
    }
  }
  B b = new B(); ...
}
```

Java

It is transformed to:

Java

```
public int m(){
  final int il = 0;
  class B{
    private int _new_field_il;

    public B(int i1){
      super();
      _new_field_il=i1;
    }

    public int m(){
      return i+_new_field_il;
    }
  }
  B b = new B(il); ...
}
```

Java

2.3.8 Sum Comprehensions

The reasoning about the memory consumption of loops as presented in Section 13 necessitates support for *sum* comprehensions in Java DL. This Section gives an overview of the syntax and semantics of *sum* comprehensions in Java DL and provides also some evidence on the degree of automation achieved by the calculus rules for *sum* comprehensions.

Other (e.g., *product*) comprehensions can be implemented analogously and are not discussed here.

Definition 2.42 (Sum Terms – Syntax) *A* sum *term has the form:*

$$sum(x;\ [l, u);\ t)$$

where

- *l, u and t are terms of type integer*
- *x is a logic integer variable bound in t (but not in l and u)*

Definition 2.43 (Sum Terms – Semantics) *Let $\mathcal{K} := (\mathcal{M}, \mathcal{S}, *, \rho)$ be a* Java DL *Kripke structure, $s \in \mathcal{S}$ and β a variable assignment. Then for an arbitrary* sum *term $sum(x; [l,u); t)$:*

$$val_{s,\beta}(sum(x;\ [l,u);\ t)) := \begin{cases} \sum\limits_{i:=val_{s,\beta}(l)}^{val_{s,\beta}(u-1)} val_{s,\beta_x^i}(t) & \textit{if } val_{s,\beta}(l) < val_{s,\beta}(u) \\ 0 & \textit{otherwise} \end{cases}$$

As definition 2.43 states, a term sum$(x;\ [l,u);\ t)$ represents the "sum of t for all values of x in the half-open (integer) interval $[u,l)$" (where the bounds u and l do not depend on x since x is only bound in t). By limiting the range of the summation to the interval $[u,l)$ we ensure that we only have to consider finite sums. Since t can be an arbitrary term of type integer, we can further filter the summands by using an *if-then-else* term here.

Using a half-open interval (instead of a closed one as the mathematical notation suggests) bears some technical advantages, as pointed out in the following.

Axioms

This Section lists the axioms we assume for sums and their representation as taclets. All additional taclets (partly defined for increasing automation, partly for convenience in interactive proof finding) presented in the following are entailed by these axioms which was also formally proven[10].

We define the semantics of sums inductively. First the base case, which states that a sum over an empty range equals zero:

Axiom 2.44 (Sum Empty) *Let a,b and $s(i)$ be integers with $b < a$. Then the following holds:*

$$\sum_{i=a}^{b} s(i) = 0$$

And its representation as a taclet:

$$\mathsf{sumEmpty}\ \frac{\begin{array}{c}\Gamma \Rightarrow u \leq l,\ \Delta \\ [\text{sum}(x;\ [l,u);\ t) \rightsquigarrow 0]\end{array}}{\Gamma \Rightarrow \Delta} \ni \text{sum}(x;\ [l,u);\ t)$$

And finally the induction step, linking the value of the sum to the values of its summands:

Axiom 2.45 (Sum Induction Upper) *Let a,b and $s(i)$ be integers with $a \leq b$. Then the following holds:*

$$(\sum_{i=a}^{b-1} s(i)) + s(b) = \sum_{i=a}^{b} s(i)$$

Which can be formulated as a taclet as follows:

$$\mathsf{sumIndU1}\quad \text{sum}(x;\ [l,u);\ t) \rightsquigarrow \text{sum}(x;\ [l,u-1);\ t) + \backslash if(l < u)\,\backslash then(t^{[x/u-1]})\,\backslash else(0)$$

[10] These proofs were performed using the approach described in [Bubel et al., 2008b, Rümmer, 2003] for verifying the correctness of taclets. The proof files can be found under the following URL: `http://i12www.ira.uka.de/~engelc/bsumTacletsAndProofs.tgz`

Derived Rules

The additional taclets derived from sumIndU1 and sumEmpty can be subdivided in two classes:

- interactive rules providing basic operations on sums (such as splitting sums in sub-sums or splitting off summands from sums) and
- automatically applicable rules normalizing sum terms (such as eliminating arithmetic operations in summands by using commutativity, distributivity and associativity of sums) and providing specialized inductive rules for sums for patterns typically occurring in induction proofs/proofs using loop invariant.

In the following some examples for automatically and solely interactively applicable rules is provided. A full account of taclets for sum terms can be found in Appendix A

Interactive Rules The rule sumSplit splits a sum in two sub-sums at an index m in case m is inside the summation range:

$$\text{sumSplit}\ \frac{\begin{array}{c}\Gamma \Rightarrow l \leq m \wedge m \leq u,\ \Delta \\ [\text{sum}(x;\ [l,u);\ t) \rightsquigarrow \text{sum}(x;\ [l,m);\ t) + \text{sum}(x;\ [m,u);\ t)]\end{array}}{\Gamma \Rightarrow \Delta} \ni \text{sum}(x;\ [l,u);\ t)$$

This taclet also shows an advantage of choosing the summation range to be a half-open interval: m can directly used as the upper and lower bound of the summation interval of the two sub-sums. A closed interval would necessitate to use either $m+1$ or $m-1$ (depending on how exactly the split rule is formulated) in one of the cases. This bears, however, the disadvantage that automatically applicable inductive taclets (to be defined in the following) match on cases where one of the bounds is increased or decreased by one, a case typically occurring in induction proofs or proofs using a loop invariant involving sum terms. The second example for a solely interactively applicable rule is the convenience rule singleSummand replacing a sum over a single summand with by this summand:

$$\text{singleSummand}\ \frac{\begin{array}{c}\Gamma \Rightarrow l \doteq u - 1,\ \Delta \\ \left[\text{sum}(x;\ [l,u);\ t) \rightsquigarrow t^{[x/l]}\right]\end{array}}{\Gamma \Rightarrow \Delta} \ni \text{sum}(x;\ [l,u);\ t)$$

Our last example is the taclet sumOneZero dealing with sums of a special form stemming from the translation of JML's `\num_of` construct to Java DL, which allows counting the number of integers satisfying a certain formula (or boolean expression in JML) b:

$$\text{sumOneZero}\ \frac{\Gamma,\ s \geq 0 \Rightarrow \Delta}{\Gamma \Rightarrow \Delta} \ni s$$

where s stands for a term of the form $\text{sum}(x;\ [l,u);\ \backslash if(b)\, \backslash then(1)\, \backslash else(0))$.

Automatically Applicable Rules Sum terms are automatically normalized to that respect that addition operations in the summand term are resolved by utilizing the commutativity and associativity of sum terms:

$$\text{sumCommutativeAssociative}\quad \text{sum}(x;\ [l,u);\ t_1 + t_2) \rightsquigarrow \text{sum}(x;\ [l,u);\ t_1) + \text{sum}(x;\ [l,u);\ t_2)$$

If the summand is of the form $t_1 * t_2$, where the logic variable x we sum over has no free occurrence in t_2, the factor t_2 can be pulled out due to the distributivity of sums:

$$\mathsf{sumDistributive} \quad \mathrm{sum}(x;\ [l,u);\ t_1 * t_2) \rightsquigarrow \mathrm{sum}(x;\ [l,u);\ t_1) * t_2$$

The backbone of our calculus extension for sum terms is formed by rules matching certain cases frequently occurring when applying integer induction or loop invariants. These rules are based on the assumption, that in induction proofs over an induction hypothesis $\phi[n]$ making some statement about a sum term $\mathrm{sum}(x;\ [l,u);\ t)$ there typically is a linear dependence between either the upper (u) or lower (l) bound and the induction variable n of the form $u = \pm n + c$ (or $l = \pm n + c$), where c is an arbitrary constant. When doing the induction step for proving the implication $\phi[n] \to \phi[n+1]$, the upper (or lower) bound (of the sum term occurring in $\phi[n+1]$) is increased/decreased by 1 (in comparison to its occurrence in $\phi[n]$) resulting in a new upper/lower bound of the form $\pm n + c \pm 1$. Proofs performed using loop invariants exhibit a similar behavior.

Therefore, the four automatically applicable rewrite rules sumIndU1Concr, sumIndU2Concr, sumIndL1Concr and sumIndL2Concr cover the four possible cases of a summation bound being in- or decreased by one:

$$\mathsf{sumIndU1Concr} \quad \mathrm{sum}(x;\ [l,1+u);\ t) \rightsquigarrow \mathrm{sum}(x;\ [l,u);\ t) + \backslash \mathit{if}(l \leq u) \backslash \mathit{then}(t^{[x/u]}) \backslash \mathit{else}(0)$$

$$\mathsf{sumIndU2Concr} \quad \mathrm{sum}(x;\ [l,-1+u);\ t) \rightsquigarrow \mathrm{sum}(x;\ [l,u);\ t) - \backslash \mathit{if}(l < u) \backslash \mathit{then}(t^{[x/u-1]}) \backslash \mathit{else}(0)$$

$$\mathsf{sumIndL1Concr} \quad \mathrm{sum}(x;\ [l-1,u);\ t) \rightsquigarrow \mathrm{sum}(x;\ [l,u);\ t) + \backslash \mathit{if}(l < u) \backslash \mathit{then}(t^{[x/l-1]}) \backslash \mathit{else}(0)$$

$$\mathsf{sumIndL2Concr} \quad \mathrm{sum}(x;\ [1+l,u);\ t) \rightsquigarrow \mathrm{sum}(x;\ [l,u);\ t) - \backslash \mathit{if}(l < u) \backslash \mathit{then}(t^{[x/l]}) \backslash \mathit{else}(0)$$

Figures 2.1 and 2.2 show four methods which can be verified (against their JML specification) each using one of the four presented rules[11]. The example programs shown in Figures 2.1–2.3 were taken from [Dijkstra and Feijen, 1988].

In the following use a self-defined JML-like notation

— JML —

```
(\bsum int i; l; u; t)
```

— JML —

that is equivalent to (T^{DL}_{JML} denotes a mapping from JML expressions to Java DL terms and formulas here)

$$\mathrm{sum}(T^{DL}_{JML}(i);\ [T^{DL}_{JML}(l), T^{DL}_{JML}(u));\ T^{DL}_{JML}(t))$$

for specifying properties over sums in JML.

All four methods compute the sum over the elements of an integer array `a` in a while loop summing up the array elements in a variable `s` with `sum0` and `sum1` starting at the beginning of the array and `sum2` and `sum3` starting at its end. The method specifications are identical in all four cases.

We shortly review the role of our above defined rules in the proof of `sum0`'s method specification. The loop invariant defined in `sum0` entails that

[11] For these examples we are applying an idealised semantics interpreting Java integers ($\mathbb{Z}_{/2^{32}\mathbb{Z}}$) as unbounded integers ($\mathbb{Z}$) which is in line with the semantics of sum terms denoting a summation in $\mathbb{Z}$. Faithfulness to the actual Java semantics would necessitate a cast **(int)** to be applied to each of the **\bsum** expressions.

Java + JML

```
/*@ public normal_behavior
  @  ensures \result==(\bsum int i; 0; a.length; a[i]);
  @*/
public static int sum0(int[] a){
  int s = 0, n = 0;
  /*@ loop_invariant n>=0 && n<=a.length && s==(\bsum int i; 0; n; a[i]);
    @ assignable s, n;
    @ decreasing a.length-n;
    @*/
  while(n < a.length){
    s += a[n++];
  }
  return s;
}

/*@ public normal_behavior
  @  ensures \result==(\bsum int i; 0; a.length; a[i]);
  @*/
public static int sum1(int[] a){
  int s = 0, n=0;
  /*@ loop_invariant n>=0 && n<=a.length &&
    @  s+(\bsum int i; n; a.length; a[i])==(\bsum int i; 0; a.length; a[i]);
    @ assignable s, n;
    @ decreasing a.length-n;
    @*/
  while(n < a.length){
    s += a[n++];
  }
  return s;
}
```

Java + JML

Figure 2.1: Two methods for summing an array's elements starting from the first element. The code and method specifications of both methods are identical, the loop invariants however differ. The contracts of both methods were proven correct automatically requiring the application of the following rules for treating sum comprehensions: in both cases the rule sumLEqU was required for showing the initial validity of the loop invariant. In addition, for proving the loop invariant to be preserved in `sum0` (`sum1`) the rule sumIndU1Concr (sumIndL2Concr) needed to be applied (since the *upper* (*lower*) bound `n` of `(\bsum int i; 0; n; a[i])` (`(\bsum int i; n; a.length; a[i])`) is increased by the loop body by 1).

Java + JML

```
/*@ public normal_behavior
  @  ensures \result==(\bsum int i; 0; a.length; a[i]);
  @*/
public static int sum2(int[] a){
  int s = 0, n = a.length;
  /*@ loop_invariant n>=0 && n<=a.length &&
    @    s==(\bsum int i; n; a.length; a[i]);
    @ assignable s, n;
    @ decreasing n;
    @*/
  while(n > 0){
    s += a[--n];
  }
  return s;
}

/*@ public normal_behavior
  @  ensures \result==(\bsum int i; 0; a.length; a[i]);
  @*/
public static int sum3(int[] a){
  int s = 0, n = a.length;
  /*@ loop_invariant n>=0 && n<=a.length &&
    @    s + (\bsum int i; 0; n; a[i])==(\bsum int i; 0; a.length; a[i]);
    @ assignable s, n;
    @ decreasing n;
    @*/
  while(n > 0){
    s += a[--n];
  }
  return s;
}
```

Java + JML

Figure 2.2: Two methods for summing an array's elements starting from the last element. The contracts of both methods were proven correct automatically requiring the application of the following rules for treating sum comprehensions: For `sum2` the rule sumLEqU was needed for showing the initial validity of the loop invariant as well as sumIndL1Concr for proving the invariant to be preserved. In the proof for `sum3`'s contract sumIndU2Concr is required for proving the preserving of the loop invariant.

— JML —
```
s==(\bsum int i; 0; n; a[i])
```
— JML —

For proving that this loop invariant is preserved we basically have to show that

$$s \doteq \mathrm{sum}(i;\ [0,n);\ a[i]) \land I' \land n < a.length \rightarrow \langle \texttt{s+=a[n++];} \rangle\, s \doteq \mathrm{sum}(i;\ [0,n);\ a[i]) \land I'$$

holds (where I' is the remainder of the invariant). This is equivalent to a formula of the form (this is determined by symbolic execution of `s+=a[n++];` in the Java DL calculus):

$$s \doteq \mathrm{sum}(i;\ [0,n);\ a[i]) \land I' \land n < a.length \rightarrow s + a[n] \doteq \mathrm{sum}(i;\ [0,n+1);\ a[i]) \land I''$$

I'' is trivially implied by $I' \land n < a.length$. What is left to prove is that $\mathrm{sum}(i;\ [0,n+1);\ a[i])$ is entailed by the premise of the implication. Here the taclet sumIndU1Concr helps us to determine how the two sum terms relate to each other:

It can be applied to $\mathrm{sum}(i;\ [0,n+1);\ a[i])$ resulting in a term

$$\mathrm{sum}(i;\ [0,n);\ a[i]) + \backslash if(0 \leq n)\,\backslash then(a[n])\,\backslash else(0)$$

which can be simplified to

$$\mathrm{sum}(i;\ [0,n);\ a[i]) + a[n]$$

leading to the formula

$$s \doteq \mathrm{sum}(i;\ [0,n);\ a[i]) \land I' \land n < a.length \rightarrow s + a[n] \doteq \mathrm{sum}(i;\ [0,n);\ a[i]) + a[n]$$

which is obviously valid and can also be proven automatically.

For showing that the loop invariant is initially (before the first loop iteration) valid we define another specialized taclet matching the case that the lower and upper bound of a sum term are (syntactically) identical:

$$\textsf{sumLEqU} \quad \mathrm{sum}(x;\ [l,l);\ t) \rightsquigarrow 0$$

In method `sum1` we have to deal with a sum term whose lower bound is increased by the loop body requiring application of sumIndL2Concr to prove its method spec. Examples for specifications requiring the use of sumIndL1Concr and sumIndU2Concr are provided in figure 2.2.

Evaluation

In [Leino and Monahan, 2007] a similar technique for handling comprehensions in the Spec# program verifier [Barnett et al., 2005] is described. The authors provide also performance measurements of the presented approach applied to several example programs taken from [Dijkstra and Feijen, 1988]. These include the already discussed programs shown in figures 2.1 and 2.2 and the more complex method `coincidenceCount1` (see figure 2.3) featuring a nested sum comprehension occurring in its loop invariant.

Table 2.2 compares the performance of an implementation of the just presented approach in the KeY system with the one from [Leino and Monahan, 2007]. The figures are not comparable one-to-one also due to different hardware, but first of all due to [Leino and Monahan, 2007]

Java + JML

```
/*@ public normal_behavior
  @  requires (\forall int i,j; 0<=i && i<j && j<f.length; f[i]<f[j]);
  @  requires (\forall int i,j; 0<=i && i<j && j<g.length; g[i]<g[j]);
  @  requires f!=null && g!=null;
  @  ensures \result==(\bsum int i; 0; f.length;
  @            (\bsum int j; 0; g.length; (f[i]==g[j]?1:0)));
  @*/
public static int coincidenceCount1(int[] f, int[] g){
  int ct=0, m=0, n=0;
  /*@ loop_invariant m>=0 && n>=0 && m<=f.length && n<=g.length &&
    @    ct ==(\bsum int i; 0; m;
    @            (\bsum int j; 0; n; (f[i]==g[j]?1:0))) &&
    @    (m==f.length || (\forall int j; j>=0 && j<n; g[j]<f[m])) &&
    @    (n==g.length || (\forall int i; i>=0 && i<m; f[i]<g[n]));
    @ assignable ct, n, m;
    @ decreasing f.length-m + g.length-n;
    @*/
  while(m<f.length && n<g.length){
    if(f[m]<g[n]){
      m++;
    }else if(g[n]<f[m]){
      n++;
    }else{
      ct++;
      m++;
      n++;
    }
  }
  return ct;
}
```

Java + JML

Figure 2.3: coincidenceCount1 annotated with JML specifications. The method contract was proven correct automatically, which involved application of sumIndU1Concr, sumZeroRight, sumCommutativeAssociative, sumEqual2, sumZero, sumEqSplit1′, sumEqSplit2′, sumEqZeroCut and sumLEqU.

Method	KeY	Spec#/Simplify	Spec#/Z3	KeY/Number of Rule Applications
sum0	0.8	0.219	0.172	502
sum1	0.9	0.063	0.016	569
sum2	0.6	0.047	0.016	418
sum3	0.9	0.110	0.016	475
coincidenceCount1	12.3	18.970	–	4884

Table 2.2: Performance Measurements (in seconds) of program verifications. The measurements for the Spec# approach were taken from [Leino and Monahan, 2007] and measured on a Core 2 Duo running at 2.33GHz with 4MB of L2 cache. The measurements for the KeY approach were taken on roughly comparable hardware: an 1.83GHz Core 2 Duo laptop with 2MB L2 cache. A dash "–" indicates that no proof was found.

only verifying partial correctness. The comparison, however, shows that not only a high level of automation (as all examples could be proven automatically) can be achieved in treating comprehensions when concentrating on handling patterns typically occurring in induction proofs (using loop invariants basically means doing an induction proof), the formalization is also efficient in terms of required rule applications and thus also execution time.

3 Java Modeling Language

The *Java Modeling Language* (JML) [Leavens et al., 2007, Leavens et al., 2006] is a behavioral interface specification language dedicated to Java, which allows specifying the functional and non-functional behavior of Java programs. This Section reviews the basic concepts of JML relevant for this work and explains how JML expressions relate to Java DL terms and formulas and JML method specifications and class invariants are translated to Java DL contracts (see Section 2.3.5).

JML specifications are either added to the targeted Java code as comments enclosed in the JML-specific comment delimiters `/*@` and `@*/` (or `//@` for single-line specifications) or listed in a separate file. The following example shows a Java class annotated with a JML class invariant and several JML method specification cases :

— Java + JML —

```
public class MyArrayList{

  int count;
  public Object[] elements;

  /*@ public invariant 0<=count && count<=elements.length &&
    @      (\forall int i; 0<=i && i<elements.length;
    @                      elements[i]!=null <==> i<count); @*/

  ...

  /*@ public normal_behavior
    @   requires !(\exists int i; 0<=i && i<count; elements[i].equals(o))
    @              && o!=null && count<elements.length;
    @   assignable elements[count], count;
    @   ensures elements[\old(count)]==o && count==\old(count)+1;
    @ also public exceptional_behavior
    @   requires o==null;
    @   assignable \nothing;
    @   signals (IllegalArgumentException) true;
    @ also ...   @*/
  public boolean add(Object /*@nullable@*/ o)
                 throws IllegalArgumentException{...}
}
```

— Java + JML —

In JML invariants and pre- and postconditions are represented by JML expressions of type boolean. As exemplified by the above class invariant, a JML expression can either be a side-effect free Java expression, such as `0<=count`, or a JML specific construct (in general starting with a \), for instance a quantified expression like:

Java + JML

```
(\forall int i; 0<=i && i<elements.length; elements[i]==null <==> i<count)
```

Java + JML

A quantified expression in JML is subdivided in three segments

- the first segment (represented by **`int`** `i` in the above example) declares the variables bound by the quantifier in the remainder of the quantified expression.
- the second segment (`0<=i && i<elements.length`) is an expression (range predicate) constraining the range of quantification to those values of the quantified variables making the range predicate true.
- a third segment (`elements[i]==`**`null`** `<==> i<count`) expressing some property over the quantified variables.

Most JML expressions can be mapped canonically to Java DL (as described in [Engel, 2005]). In the following we denote this mapping with T^{DL}_{JML}. The above quantified expression is translated to a Java DL formula as follows:

$$T^{DL}_{JML}(\texttt{(\textbackslash\textbf{forall} \textbf{int} i; 0<=i\&\&i<elements.length; elements[i]!=\textbf{null}<==>i<count)})$$
$$:=$$
$$\forall\ \mathit{int}\ i; (0 \leq i \land i < \mathit{elements.length}) \rightarrow (\mathit{elements}[i] \not\doteq \texttt{null} \leftrightarrow i < \mathit{count})$$

As before, we use $s \not\doteq t$ as syntactic sugar for $\neg s \doteq t$.

The method specification of method `add` is structured in several specification cases (or *spec cases* for short) concatenated by the **`also`** keyword. Each specification case consists of (we only consider a selection of clauses relevant in the following)

- implicitly conjoint **`requires`** clauses of type **`boolean`** specifying the precondition.
- an **`assignable`** clause (see Section 9.2 for details) consisting of JML location descriptors describing the set of locations the method can assign to.
- implicitly conjoint **`ensures`** clauses of type **`boolean`** specifying the postcondition in case of normal termination.
- implicitly conjoint **`signals`** clauses of type **`boolean`** specifying the postcondition in case of abrupt termination (by raising an exception). The signals clause also allows to specify the type of the exception and the respective postcondition that has to hold in case an exception of this type is raised by the method.
- a **`depends`** clause (see Section 9.2 for details) consisting of JML location descriptors describing the set of locations the method's behavior depends on.
- a **`captures`** clause (see Section 9.2 for details) consisting of reference-type JML location descriptors describing the set of locations whose values may be captured (i.e., fresh references to this values outliving the method's execution may be created) by the method.

If any of the above clauses is not explicitly specified by the programmer the most permissive value is taken as default, e.g., **`true`** in case of a pre- or postcondition.

In the general case JML method spec cases start with the **`behavior`** keyword. JML defines two additional types of spec cases:

- the **`normal_behavior`** spec case indicating normal termination of the specified method in case the precondition of the spec case is met. A **`normal_behavior`** can be desugared to a **`behavior`** spec case by adding the clause

  — Java + JML —

  ```
  signals (Exception) false;
  ```

  — Java + JML —

- the **`exceptional_behavior`** spec case indicating abrupt termination of the specified method in case the precondition of the spec case is met. Again this spec case is just syntactic sugar for a **`behavior`** spec case containing the clause

  — Java + JML —

  ```
  ensures false;
  ```

  — Java + JML —

Thus the **`normal_behavior`** spec case of `add` corresponds to the Java DL contract (see Section 2.3.5):

$$\begin{array}{l}(\neg\exists\ \mathit{int}\ i; (0 < i \land i \leq \mathit{count} \rightarrow \mathit{elements}[i].\mathit{equals}(o)) \land o \not\doteq \mathtt{null} \land \mathit{count} < \mathit{elements.length},\\ \ (\mathit{exc} \doteq \mathtt{null} \rightarrow \mathit{post}_{\mathit{normal}}) \land (\mathit{exc} \not\doteq \mathtt{null}\ \rightarrow \mathit{post}_{\mathit{exceptional}}),\\ \ \{\mathit{elements}[\mathit{count}],\ \mathit{count}\},\\ \ *,\\ \ *)\end{array}$$

where

$$\mathit{post}_{\mathit{normal}} := \mathit{elements}[\mathit{count}^{@pre}] \doteq o \land \mathit{count} \doteq \mathit{count}^{@pre} + 1$$

and

$$\mathit{post}_{\mathit{exceptional}} := \mathit{instanceof}_{\mathit{Exception}}(\mathit{exc}) \doteq \mathit{TRUE} \rightarrow \mathit{false}$$

with $\mathit{instanceof}_T(\mathit{exc})$ for any type T are rigid function symbols with

$$\mathit{val}_{s,\beta}(\mathit{instanceof}_T(t)) := \begin{cases} \mathit{val}_s(\mathit{TRUE}) & \text{if } \delta(\mathit{val}_{s,\beta}(t)) \sqsubseteq T \\ \mathit{val}_s(\mathit{FALSE}) & \text{otherwise} \end{cases}$$

Since $\mathit{post}_{\mathit{exceptional}}$ is logically equivalent to false, the postcondition

$$(\mathit{exc} \doteq \mathtt{null} \rightarrow \mathit{post}_{\mathit{normal}}) \land (\mathit{exc} \not\doteq \mathtt{null}\ \rightarrow \mathit{post}_{\mathit{exceptional}})$$

is equivalent to

$$\mathit{post}_{\mathit{normal}} \land \mathit{exc} \doteq \mathtt{null}$$

The `\`**`old`** construct occurring in the ensures clause evaluates the expression it is applied to in the pre-state of the method. We therefore get $T^{DL}_{JML}(\backslash\mathtt{old(count)}) = \mathit{count}^{@pre}$. The

captures and the **depends** clause are not specified so their default value is taken which is $*$ (the entire heap).

The **exceptional_behavior** spec case yields the JAVA DL contract:

$$\begin{array}{l}(o \doteq \mathtt{null},\\ (exc \doteq \mathtt{null} \rightarrow post_{normal}) \wedge (exc \not\doteq \mathtt{null} \rightarrow post_{exceptional}),\\ \emptyset,\\ *,\\ *)\end{array}$$

where

$$post_{normal} := false$$

and

$$post_{exceptional} := instanceof_{IllegalArgumentException}(exc) \doteq TRUE \rightarrow true$$

In this case the postcondition is logically equivalent to

$$exc \not\doteq \mathtt{null}$$

Remark 3.1 (JML Assignable Clauses and Object Creation) *Assignable clauses are evaluated in the pre-state. One consequence of this is that locations of objects newly created by the specified method need not be mentioned in the assignable clause since they did not exist in the pre-state (and there is no means for referencing not yet created objects in JML). Due to the* constant domain assumption *(see 2.3.3), this approach cannot be adopted by* JAVA DL *and all locations changed by the method also those belonging to in the pre-state not yet created objects need to be listed by the assignable clause (this includes implicit fields such as* `<c>` *and* `<ntc>`*).*

*For writing JML contracts that are also correct under the constant domain assumption the construct \**object_creation**(T) was introduced which can occur in the assignable clause and denotes the set of locations potentially changed when an object of type* `T` *is created (i.e., implicit fields affected by object creation and fields of the created object or array slots of the created array).*

4 Real-Time Specification for Java

The *Real-Time Specification for Java* (RTSJ) [Bollella and Gosling, 2000] is the result of the *Java Specification Request 1* (JSR001, see also `http://jcp.org`) aiming at making Java suitable for real-time applications. RTSJ addresses several issues important for real-time applications which were not satisfyingly (from the real-time programming perspective) handled by standard Java so far including scheduling, event-handling and memory management. This Section shortly reviews the novel memory model (MM) introduced by the RTSJ.

4.1 Memory Management

A garbage collector [Jones and Lins, 1996] as employed by Java carries numerous advantages: it helps, for instance, to prevent memory leaks, eliminates the possibility of creating dangling references (a dangling reference is a reference to an object that has already been reclaimed) and reduces the programming effort since objects no longer in use do not have to be deallocated explicitly by the programmer. For real-time programming, however, garbage collection poses a severe problem (even though this is mitigated by real-time garbage collectors [Siebert, 2007] to a certain extend): It introduces indeterministic performance, since the garbage collector thread can interrupt other threads at arbitrary times and for unbounded periods of time.

4.1.1 Memory Areas

The RTSJ approaches this problem by defining a region-based memory model (region-based memory management is not an RTSJ-specific concept, [Tofte and Talpin, 1997], for instance, proposes a region-based memory model for ML) featuring, in addition to the classical Java heap memory, two novel kinds of memory regions that are not subject to garbage collection: *immortal memory* and *scoped memory*. Both are represented by Java classes; the RTSJ entails no changes to Java on the syntax level.

An immortal memory area, which is always a singleton, is never garbage collected and never freed during the lifetime of the application. In contrast, a scoped memory area (or just: scope), of which arbitrarily many can be created by an application, is freed at well defined occasions, namely as soon as no thread is active inside it any more. As, for an application, scopes are just represented by "ordinary" objects, new scopes can be created via the **`new`** operator at runtime. Heap memory is in principle also usable by RTSJ-based applications which is for afore mentioned reasons not advisable in a hard real-time context. This work will thus only consider programs running exclusively in immortal and scoped memory.

Let us now review how scopes are used in RTSJ programs and fix the nomenclature for talking about operations on (and relations between) scopes. We start with an overview of the different kinds of memory areas and the corresponding API classes before describing the concept of the scope stack and how it is affected by RTSJ API methods.

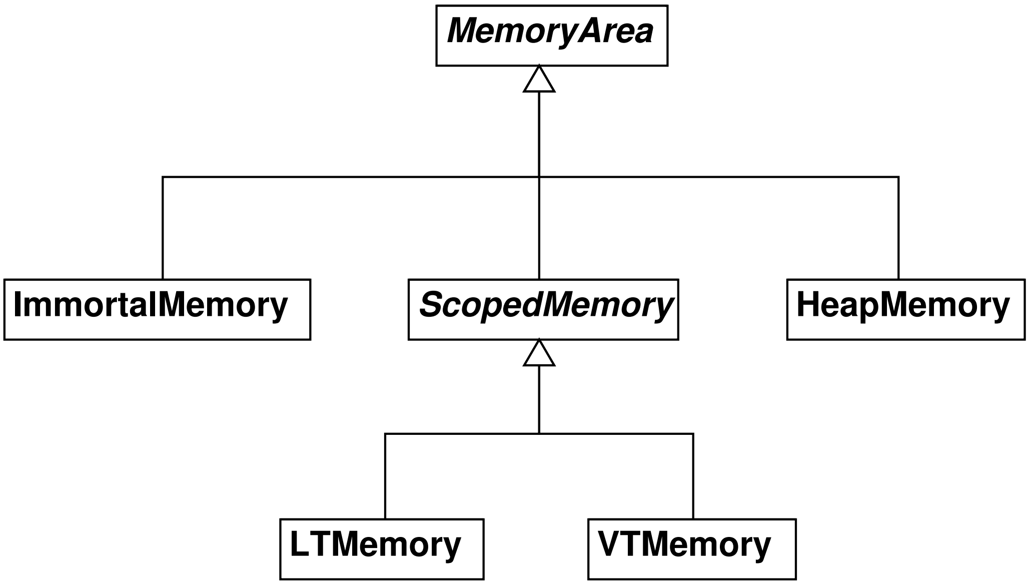


Figure 4.1: RTSJ memory area class hierarchy (abstract classes are printed italic)

Class Hierarchy

The various kinds of memory areas provided by the RTSJ are each represented by a JAVA class extending the abstract class `javax.realtime.MemoryArea`[1]. This applies also to the standard JAVA heap memory which is given by a singleton instance of the class `HeapMemory` extending `MemoryArea`.

Figure 4.1 shows the class hierarchy of memory areas in the RTSJ. *Scoped Memory* is represented by the abstract class `ScopedMemory`. The RTSJ provides two classes implementing `ScopedMemory`:

- `LTMemory`: a scoped memory area where allocations take a linear (relative to the size of the allocated object) amount of time.

- `VTMemory`: a scoped memory area without a linear correspondence between object size and allocation time.

Due to its more deterministic performance characteristics, `LTMemory` is preferable for hard real-time and safety-critical [Kwon et al., 2005, Kung et al., 2006] applications.

The Current Scope

For each thread and each program state the allocation context (i.e., memory area) from which a thread allocates memory via the **`new`** operator is uniquely determined. We call this allocation context the thread's *current scope* in the following. The current scope may change during program execution as threads can enter and leave scopes.

[1] For the sake of readability package prefixes for frequently used classes are often omitted in the following.

Entering a Scope

A thread can enter a scope which has the effect the entered scope becomes the new *current scope* of the thread entering it. As already mentioned, scopes are represented by objects of type `MemoryArea`. From a technical point of view, entering a scope is performed by calling the method `enter@MemoryArea` which takes an object of type `java.lang.Runnable` as argument. The `run@Runnable` method of the passed `Runnable` object is then executed in this new current scope. When the `enter` method terminates, the scope which was the current scope before execution of the `enter` method becomes current scope again.

Nesting of Scopes and the Scope Stack

When a scope b was entered from a scope a, we say b is an inner scope of a. Accordingly, we call a an outer scope of b. The scope b remains an inner scope of a until all invocations (for all threads) of `enter@MemoryArea` invoked from an execution context in which a was the current scope and having b as receiver object have terminated. As nesting constitutes a transitive relation, we denote also inner nested scopes as inner and outer nested scopes as outer scopes. Inner scopes must always be left before their respective outer scopes[2]. The nesting of scopes can therefore be described by a stack. In the following we depict the nesting of scopes as an upward growing stack (inner scopes are located above outer scopes on this stack).

The Method `executeInArea` and the Cactus Stack

Besides the method `enter`, the RTSJ offers a second possibility to change a threads current scope: the method `executeInArea@MemoryArea` that, as the method `enter`, takes a `Runnable` object as argument. When `executeInArea` is invoked on a receiver object a, that is an outer scope of the current scope (if this constraint is not met, an exception is raised) a becomes the current scope when the `run` method of the passed `Runnable` object is executed. After `executeInArea` has terminated the previous current scope becomes current scope again. Note, that invoking `executeInArea` does not change the scope stack. There is, however, an indirect effect the execution of `executeInArea` can have on the structure of the scope stack: during the execution of `executeInArea` other scopes may be entered. This can either concern a scope that is already a direct inner scope of the current scope (thus entering this scope does not change the scope stack) or a scope that is not yet located on the scope stack. Due to this, every scope can have several direct inner scopes which makes the scope stack a cactus stack.

Remark 4.1 (Notation Used for Memory Stacks) *On several occasions, we will in the following visualize scope stacks by diagrams for which we use the following notations: A single scope `s1` with several objects allocated in it is depicted as in Figure 4.2. The topmost line contains a unique identifier for the scope, the area below unique identifiers for (a selection of) objects allocated in this scope. If we are just interested in the structure of the scope stack and not in any objects allocated in a scope we use the simpler notation shown in Figure 4.3.*

Scope stacks grow upward in our notation. Figure 4.4 shows a stack in which two scopes

[2] This obviously holds, since on the call stack, the invocation of `enter` corresponding to the inner scope is located above the outer scope's corresponding method invocation.

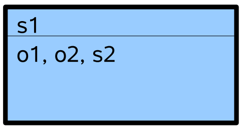


Figure 4.2: A scope `s1` containing objects `o1`, `o2`, `s2`

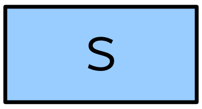


Figure 4.3: A scope `S`

`s1` and `s2` are allocated in immortal memory and then subsequently entered. Such a stack is created if the following piece of code is executed in immortal memory:

— *Java* —

```java
final ScopedMemory s1 = new LTMemory(10000);
final ScopedMemory s2 = new LTMemory(10000);
s1.enter(new Runnable(){
  public void run(){
    s2.enter(new Runnable(){
      public void run(){
        ...
```

— *Java* —

The two `Runnable` objects created in the above code are denoted as `r1` and `r2` in Figure 4.4.

Example 4.2 *Figure 4.5 shows what effects the execution of the `enter` and `executeInArea` method has on the scope stack and the current scope:*

- *We start in the current scope `s2`*
- *from which we call `s3.enter(...)` which pushes `s3` on the scope stack above `s2` and makes `s3` the new current scope.*

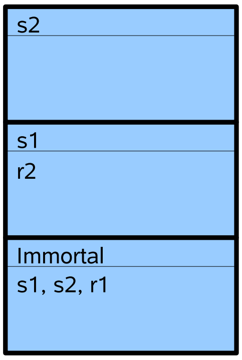


Figure 4.4: A scope stack

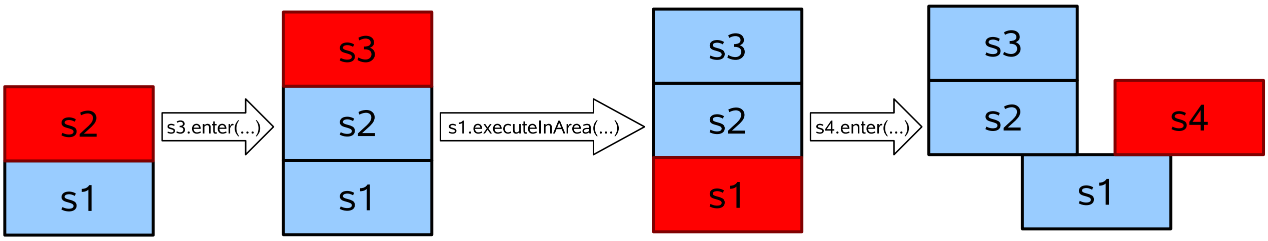


Figure 4.5: Effects of `enter` and `executeInArea`. The current scope is marked red.

- *While `s3` is still the current scope (`s3.enter(...)` has not yet terminated) we call `s1.executeInArea(...)`, with `s1` being an outer scope of the current scope `s3`. Thus `s1` becomes the new current scope.*
- *While `s1` is still the current scope we execute `s4.enter(...)`. Thus `s4` is pushed on the scope stack above `s1` (or, in other words: becomes a direct inner scope of `s1`). This leads to a branching of the scope stack as `s1` has now two direct inner scopes: `s2` and `s4`.*

Real-Time Threads

For leveraging the properties of its novel kinds of memory areas the RTSJ defines the class `NoHeapRealtimeThread` (NHRTT) using exclusively scoped and immortal memory. Therefore, a NHRTT does not interfere with the garbage collector, which operates solely on the heap, and can thus interrupt the garbage collector at arbitrary occasions.

4.1.2 Assignment Checks

When assigning to locations of non-primitive type (that are no local variables), runtime checks are performed to prevent the creation of such references that can potentially give rise to dangling references. A failed check raises an `IllegalAssignmentError`. In the following we call assignments not raising an `IllegalAssignmentError` legal and otherwise illegal. These checks enforce that for each reference `x.r` or `x[i]` to an object `y`

- `y` resides in immortal memory or
- `y` resides in a scope s_1 and `x` resides in the same or an inner scope s_2 of s_1 (s_2 is located above s_1 on at least one branch of the scope stack[3]).

In short, the creation of references outliving the referenced object is avoided by this approach. Table 4.1 gives an overview of the criteria for the admissibility of references applied by the runtime assignment checks. Heap memory, which is in this respect treated equivalently to immortal memory (as it is, like immortal memory, never freed explicitly), is not listed in this table as it is considered irrelevant for safety critical applications.

Example 4.3 (Dangling Reference) *The following piece of code would create a dangling reference as the object `o.a` refers to is reclaimed after termination of `enter`:*

[3] The *single parent rule* (see Section 4.1.3) ensures that if s_2 is located above s_1 on one branch, s_2 is located above s_1 on every branch s_2 occurs on.

From	*Reference to Immortal Memory*	*Reference to Scoped Memory*
Immortal Memory	allowed	forbidden
Scoped Memory	allowed	allowed iff. reference points to outer nested scope
Local Variable	allowed	allowed
Static Field	allowed	forbidden

Table 4.1: Constraints on References in RTSJ

Java

```
1  ScopedMemory s = new LTMemory();
2  final MyObject o = new MyObject();
3  s.enter(new Runnable(){
4    public void run(){
5      o.a = new Object();
6    }
7  }
```

Java

Therefore, the assignment in line 5 raises an `IllegalAssignmentError`*.*

4.1.3 Single Parent Rule

RTSJ also imposes certain well-formedness constraints on the scope stack which are also checked at runtime. The so-called *single parent rule* ensures that each occurrence of a scope s_1 on the scope (cactus) stack has the same parent s_2. Since this also holds for s_2 and all preceding scopes as well, the entire branch below s_1 is identical for each occurrence (on the stack) of s_1. The single parent rule is enforced when scopes are entered. A failed check again raises an exception leading to abrupt termination of the `enter` method.

The single parent rule is needed to avoid cycles in the scopes' nesting hierarchy which would render the assignment checks as described in Section 4.1.2 insufficient for preventing dangling references.

4.1.4 Portals

For facilitating sharing of data in scoped memory the RTSJ provides so-called *portals*. Each object allocated in scoped memory can serve as portal to the scope it is allocated in. Associating a scope with a portal is done by the method `setPortal@(ScopedMemory)` obtaining the portal from a certain scope is possible through the method `getPortal@(ScopedMemory)`. Both methods can give rise to an `IllegalAssignmentError` if

- (in the case of `setPortal`) the portal object is not allocated in the scope given by the receiver object of the invocation of `setPortal` or
- (in the case of `getPortal`) a reference from the context in which `getPortal` is invoked to the portal object would be illegal.

4.2 Safety-Critical Java Profiles Based on the RTSJ

The definition of a Java standard for safety critical systems is currently in progress under the Java community process (JSR302). This standard aims at subsetting the RTSJ for the sake of more deterministic behavior and better analysability.

There are several (e.g., [Schoeberl et al., 2007, Kung et al., 2006, Kwon et al., 2005]) competing proposals for the specification of such a safety-critical Java (SCJ) profile. One aspect these SCJ profiles have in common is the distinction between an (non-real-time) *initialisation* and a (real-time) *mission* phase. During the initialisation phase an initialiser thread creates objects having application lifetime as well as all threads executed in the mission phase. Also all required static initialisation is performed during the initialisation phase. During the mission phase only allocations in scopes freed after the mission phase are allowed. By this, it is ensured that the system returns to its initial state after each mission phase and can reenter its next mission phase without being newly set up (by an initialiser thread) again.

Another similarity between these profiles is that the use of scoped memory is strictly limited. The *Ravenscar Profile* [Kwon et al., 2005], for instance, permits only one scope per thread running in the mission phase. In addition, all scopes need to be created during the initialisation phase. In [Schoeberl et al., 2007] a similar restriction is imposed: Each thread is associated with one scope for dynamic allocation[4].

4.3 Verification Challenges for RTSJ Programs and Scope of This Work

The correctness of an RTSJ program can be considered from several perspectives. Functional correctness meaning the compliance of the program with a functional specification must be ensured. Functional correctness can be addressed, as also for “classical” Java, by deductive verification or static analysis. Using existing verification tools (such as [Beckert et al., 2007, Cok and Kiniry, 2004]) for sequential Java for this is, however, not possible without modification due to the different memory model and the additional runtime checks an RTSJ compliant JVM performs. Section 6 elaborates on how a reasonably restricted version of the RTSJ memory model can be formalized to be efficiently usable in a theorem prover for dynamic logic. The described approach has been implemented in the KeY [Beckert et al., 2007] system and was evaluated on several non-trivial examples.

As opposed to non-real-time systems, in real-time system not only functional but also non-functional properties such as *Worst-Case Execution Time* (WCET) and *Worst-Case Memory Usage* (WCMU) are an issue of correctness: A real-time system missing its deadline is incorrect irrespective of its functional behavior. As resources in real-time systems are inherently limited, it is vital to have precise and (provably) correct WCMU estimations for RT applications. This is not only important when memory is freed explicitly as it is the case for scoped memory but also when employing a real-time garbage collector. For determining the CPU utilization of such a garbage collector it is necessary to know the allocation rate of the application it is applied to [Mann et al., 2005] which is in turn a figure every schedulability analysis for determining whether the regarded system meets its deadline depends on. WCMU contracts

[4] The authors of [Schoeberl et al., 2007] consider this an intermediate solution and propose compiler generated scopes as future work.

can obviously add to determining such an allocation rate correctly.

Sections 11–17 describe a way to modularly specify and verify WCMU contracts. Besides being more hardware dependent, WCET time contracts [Fredriksson et al., 2007] could be formally verified analogously, but this is not in the scope of this work.

Concurrency in RT applications gives rise to new phenomena and sources for incorrectness. Accessing shared resources and data can lead to, for instance, deadlocks or starvation of threads. Concurrent read and write accesses to data can lead to race conditions or leave data in an inconsistent state. Verification results obtained for a sequential programs can be rendered incorrect when this program is employed in a concurrent setting with other threads, due to these threads possibly *interfering* with its execution. Proving *non-interference* and correctness of concurrent Java programs [Klebanov, 2004, Beckert and Klebanov, 2007] for an arbitrary number of threads is a complex endeavor since every possible (modulo symmetries) intermixing of execution progress of the threads has to be considered. Section 9 elaborates on how the RTSJ memory model can be employed for achieving improved data encapsulation guarantees which eases verification of non-interference. It also proposes proof obligations for *depends* and *captures* clauses.

Knowing about non-interference is also relevant when it comes to real-time model checking (for instance [Bengtsson et al., 1996]) which can be applied to check safety and liveness properties. Code blocks known to show no interference with other threads can be considered atomic and can thus add to reducing the complexity of the model.

Another issue arising for concurrent real-time systems is scheduling: To be correct, a real-time system must possess a schedule ensuring that all deadlines are met (such a schedule is called *feasible*). The class `javax.realtime.Scheduler` responsible for scheduling of RTSJ applications incorporates a built-in feasibility analysis. However, this can only guarantee feasible schedules if the characteristics of each schedulable object (such as WCET, which again depends to a certain extent on WCMU estimations) communicated to it are correct.

Part II

Safety Critical Java

5 The Considered SCJ Profile

In this work we will not consider the RTSJ in its full complexity but only a sequential subset of it with a restricted memory model which is referred to as KeYSCJ in the following.

5.1 Constraints on the Scope Stack

KeYSCJ compliant applications are only permitted to use *immortal* and *scoped* memory (but not *heap* memory). The initial memory area, in which a KeYSCJ application starts its execution, is immortal memory. By making this restriction we implicitly forbid (classical[1]) JAVA threads which must not execute in a `javax.realtime.MemoryArea`.

We also forbid the reentering of immortal memory using its `enter` method; accessing it via `executeInArea` is permitted however. This ensures that the only place the immortal memory area can occur on the scope stack is its root. Even though this rules out legal RTSJ programs it is much less restrictive than *safety critical Java* profiles imposing similar restrictions [Schoeberl et al., 2007, Kwon et al., 2005, HIJA, 2006].

5.2 Finalization and Deletion of Objects

Finalization of objects is not explicitly taken into account by the calculus. This means in detail: After the reference count of a scoped memory area drops to 0 the finalizers of the objects contained in this area are *not* (symbolically) executed. Also, we consider objects allocated in a reclaimed memory scope to be still created after freeing the scope. In particular this means that even though the memory in a scoped memory is reclaimed the objects contained in it are still considered to be created (as the <c> attribute is not changed). We, however, consider these objects to be moved to a specific scope containing all deleted objects (see Section 6.3.2).

To make this a sound approach we have to limit the effects a finalize method can have. A `finalize` method is not allowed to

- create objects,
- reenter the scope the finalized object is allocated in and to
- have side effects that are not local to the scope containing the finalized object. This means that for every semantic location $o.a$ or $arr[i]$ in `finalize's` *assignable* clause, the objects o or arr reside in the same scope as the finalized object.

This ensures that the execution of a `finalize` method can have no influence on the (functional) behavior of the program (performance wise, it can of course have an influence however). As objects residing in a scope whose memory was reclaimed can obviously no longer

[1] Threads which are not instances of type `javax.realtime.RealtimeThread`.

be accessed by the application, ignoring the execution of `finalize` is a viable approach for modeling object deletion: The side effects of the `finalize` method as restricted above are not observable for the code following its execution and do thus not need to be taken into account.

These constraints are expressible as, for instance, JML specifications, and could be verified formally. A formal verification of the finalize method is, however, only required in rare cases, namely in those in which it is overridden by a class. The `finalize` method defined in `java.lang.Object` is known to comply with the above restrictions.

5.3 Static Initialisation

The Java DL calculus of the KeY system is capable of taking into account the effects of static initialisation. The price to pay for this are (for most realistic programs) much more complex proofs (compared to the approach of just assuming that each class occurring in the code to be verified is already initialised). To limit the effects of static initialisation and by this ease the analysis of KeYSCJ applications we allow

- static primitive type fields only to be initialised with compile time constants and
- the static initialisation of reference type static fields to have no other side effects than on the initialised class itself.

These properties can be checked locally for each class[2]. In addition we assume that all static initialisation of classes is done in the initialisation phase of the considered RT system and no static initialisation can thus happen in the mission phase.

This allows, when performing symbolic execution, to ignore static initialisation and treat every class as already initialised since the execution of static initialisation is known to have no side effects on anything else than the initialised class itself.

Even though constraining initialisers of static fields seems to be a substantial restriction for classical Java it is only a moderate restriction for RTSJ for several reasons:

- In RT systems typically no dynamic allocation is performed. All required data structures as well as classes are initialised during an initialisation phase which leads to more deterministic performance in the mission phase.
- Besides this argument holding for RT systems in general there are special risks concerning static initialisation in RTSJ that can render a program erroneous:
  - Reference type static fields must be initialised with **`null`** or objects residing *not* in scoped memory. Thus there is the danger of failed runtime checks when a class is initialised when executing in scoped memory.
  - Object creation performed during static initialisation may give rise to *OutOfMemoryErrors* when executing in scoped memory.

For the above reasons static initialisation performed in an unrestricted way would bear a high risk for RTSJ programs.

[2] In Java DL static initialisation is modeled by an implicit method. The constraints on static initialisation concerning reference type fields can be seen as a contract for this method and be verified analogously to user-provided contracts. For primitive type fields a simple syntactical check is sufficient to determine whether they are initialised by compile time constants.

6 Calculus

This section elaborates on the handling of the RTSJ memory model in Java DL. Section 6.1 first gives a high-level overview of our approach and the changes applied to Java DL for realizing it. Section 6.3 explains in detail how we model the scope stack and operations affecting it. Taclets axiomatizing the semantics of constructs introduced in Section 6.3 are defined in Section 6.4. Section 6.5 shows how object deletion is handled in our approach. The newly defined axioms have to be considered when proving the reachability of a program state which is summarised in Section 6.6. Finally, in Section 6.7, we present rules for symbolic execution of RTSJ programs taking into account runtime checks for illegal references.

6.1 Basic Ideas

We now give a high-level and informal summary of our modeling of the RTSJ memory model. The extensions to the existing Java DL calculus required to support the scoped memory model can be subdivided as follows:

1. As for other parts of the execution context, we use the *method frame* statement to determine the allocation context (i.e., the current scope). This is done by introducing a pointer `<cma>` whose value is stored by the method frame (Section 6.2.1). Each object is associated with the scope it is allocated in via the implicit field `<ma>` (Section 6.2.2).

2. The scope stack is represented by a Java (model) class whose semantics is defined with the help of the binary predicate $\preceq$, which reflects the nesting relation of scopes, and the unary predicate im (Section 6.3.1).

3. The semantics of the relevant parts of the RTSJ API needs to be formalized which is done by JML specifications of the respective API classes and, in cases providing a sufficiently strong JML specification is not feasible[1], by a Java implementation[2] of several API methods which complies with the (informal) RTSJ API specification. This implementation (which we refer to as our reference implementation of the RTSJ API in the following) can then, like other parts of the respective code we reason about, be symbolically executed during proof finding.

4. Axioms holding in reachable (RTSJ) program states are encoded in taclets (Section 6.4). This new axioms must also be taken into account when proving that a state is reachable by execution of an admissible RTSJ program (Section 6.6).

[1] An example for such a method for which a "useful" JML specification is infeasible is `enter@MemoryArea`. Besides performing some checks `enter`'s main functionality is to invoke the `run@Runnable` method of the `Runnable` object passed as argument. Thus, its behavior is mainly determined by that of the `run` method which can, however, not be used in JML specifications (since it is, in general, not side-effect-free).

[2] This implementation is not written in pure Java but incorporates Java DL specific constructs such as the `<cma>` pointer.

5. Deleted objects are distinguished from not-deleted objects by putting them in a distinct scope holding all deleted objects. When moving objects into this scope, we need to take care that existing (legal) references are not rendered illegal (Section 6.4).
6. Calculus rules for symbolic execution of RTSJ programs are defined which reflect the RTSJ runtime checks performed upon assignments to reference type attributes and array slots (Section 6.7).

6.2 Determination of the Allocation Context

In this section we consider how the scope is determined in which an object is allocated. First (Section 6.2.1), we explain how we keep track of the *current scope*. Then, in Section 6.2.2, we show how an object is associated with the scope is allocated in.

6.2.1 The `<cma>` Pointer

We augment the Java syntax with a pointer `<cma>` (shorthand for `current memory area`) of type `MemoryArea` which points to the current scope. It is not only similar to a **`this`** pointer in this respect, it is also technically handled in a comparable way.

As described in Section 2.3.6 we use a *method frame* statement to keep track of the execution context of a method. This *method frame* also determines, for instance, to which object the **`this`** pointer is to be resolved. We enrich *method frame* to store also the memory area which `<cma>` refers to:

Java
```
method-frame(result->retvar,
             source=T,
             this=self,
             <cma>=mem) : {body}
```
Java

Except for the `enter` and `executeInArea` methods the active memory area does not change when a method is called. In this case `<cma>` is set to the value the `<cma>` pointer of the enclosing *method frame* evaluates to at the point the method call occurs which is from a technical point of view just the object `mem` stored in the enclosing *method frame*.

For describing the changeover of the memory area taking place when the `executeInArea` or `enter` method is called we define an implicit method `<runRunnable>` in class `MemoryArea` which is called from within the `enter` or `executeInArea` method respectively and triggers the memory area switch before executing the `Runnable` object `logic` passed to this memory area. When expanding a `<runRunnable>` method body statement, we therefore have to set `<cma>` to the receiver object of the method call (as for the rule methodBodyExpand in Section 2.3.6, `body` denotes the method body of the expanded method body statement):

$$\mathsf{expandRR}\ \frac{\begin{array}{l}\Gamma \Rightarrow \{\mathcal{U}\}\langle\pi\ \texttt{method-frame(}\ \texttt{result->lhs,} \\ \qquad\qquad\qquad\qquad\qquad\quad \texttt{source=MemoryArea,} \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{this}\texttt{=se,} \\ \qquad\qquad\qquad\qquad\qquad\quad \texttt{<cma>=se)}\ :\ \{\texttt{body}\}\ \omega\rangle\phi,\ \Delta\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle\pi\ \texttt{lhs=se.<runRunnable>(logic)@(MemoryArea);}\ \omega\rangle\ \phi,\ \Delta}$$

Default Memory Area

Initially, in cases no enclosing *method frame* exists, `<cma>` refers to the program variable

$$defaultMemoryArea$$

of static type `javax.realtime.MemoryArea`. In method contracts $defaultMemoryArea$ can be used as well to denote the memory area in which the specified method is called. When applying the contract as demonstrated by rule methodContractInst in Section 2.3.5, the placeholder $defaultMemoryArea$ has to be updated with the value `<cma>` resolves to at the point the considered method is called.

In this respect, the handling of `<cma>` is again similar to that of the **`this`** pointer: In method contracts we use the program variable $self$ to denote the object the **`this`** pointer of the specified method refers to. When the contract is applied, $self$ is updated with the receiver object of the method invocation (see Section 2.3.5).

6.2.2 The Implicit Field <ma>

We declare an implicit field <ma> in class `java.lang.Object` for storing the memory scope the object is allocated in. In JML specifications we access this field via

`\memoryArea``(object)`

which can be mapped to Java DL canonically:

$$T^{DL}_{JML}(\texttt{\textbackslash memoryArea(o)}) := o.\texttt{<ma>}$$

In our model, every created object is allocated in some memory area. This is also true for objects previously allocated in memory areas which have already been freed since we do not model deletion of objects. Therefore, the following axiom holds in every *reachable* state:

$$\forall\, Object\ o;\ o.\texttt{<c>} \rightarrow o.\texttt{<ma>} \neq \texttt{null} \tag{6.1}$$

We make use of this axiom in rule maNonNull:

$$\textsf{maNonNull}\ \frac{\Gamma,\ o.\texttt{<c>} \doteq TRUE,\ \mathrm{RS} \Rightarrow o.\texttt{<ma>} \doteq \texttt{null},\ \Delta}{\Gamma,\ o.\texttt{<c>} \doteq TRUE,\ \mathrm{RS} \Rightarrow \Delta}$$

The initialisation of <ma> is done by the rule allocate which is shown in its basic form in Section 2.3.3. We extend this rule here by an update setting <ma> to the memory area `<cma>` refers to at the point `<allocate>` is called:

$$\textsf{allocate}\ \frac{\begin{array}{l}\Gamma \Rightarrow \{\mathcal{U}\}\{v := get_T(T.\texttt{<ntc>})\ \|\\ \qquad T.\texttt{<ntc>} := T.\texttt{<ntc>} + 1\ \|\\ \qquad get_T(T.\texttt{<ntc>}).\texttt{<c>} := TRUE\ \|\\ \qquad get_T(T.\texttt{<ntc>}).\texttt{<ma>} := m\}\langle\pi\,\omega\rangle\,\phi,\ \Delta\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle\pi\ \texttt{v=T.<allocate>();}\ \omega\rangle\,\phi,\ \Delta}$$

Where m is the object `<cma>` resolves to, which is determined by the innermost *method frame* occurring in π.

6.3 The RTSJ API and the Scope Stack

We model the scope stack by immutable instances of the JAVA class `MemoryStack`. Each of these instances only represents a (*local*) sub branch of the entire (global) cactus stack and is as such just a "normal" stack. Each scope is augmented with an attribute `stack@MemoryStack` (which we just denote as **`stack`** in the following) representing the subbranch of the cactus stack ending with the considered memory scope and starting with the immortal memory at the root of the stack. This is possible, as for every scope s the branch below each occurrence of s on the cactus stack is identical (see Sect. 4). If **`stack`** is **`null`** the scope is not located on the cactus stack.

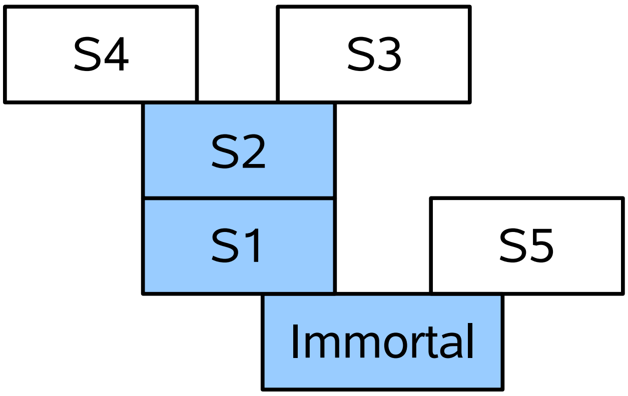


Figure 6.1: `S2.stack` represents the substack consisting of `Immortal`, `S1` and `S2`

Whenever a new scope, that is not yet located on the cactus stack, is added to it (by execution of the scope's `enter` method) its stack is initialised with the stack created by pushing the newly entered scope on the stack of the currently active memory area (the memory area from which it was entered). By doing this the local stack of the currently active memory area (as well as the stacks of all other scopes on the global stack) is not changed since the instances of `MemoryStack` are immutable.

6.3.1 The Class MemoryStack

We now need a specification for class `MemoryStack` that is sufficiently expressive but also simple enough to be efficiently used in a verification system. Simplicity of our formalization of `MemoryStack` is crucial since we do not only need to reason over the structure of the scope stack when we call methods on an instance of `MemoryArea` but each time we perform an assignment to an instance attribute or an array slot. In the following we present two alternative approaches to do this and discuss their advantages and drawbacks. First we describe an explicit modeling of the stack by means of arrays and then a lightweight modeling via imposing an ordering relation on the set of stacks.

Modelling with Arrays

Since `MemoryStack` must only represent a non-branching stack, it is possible to model it by means of JAVA arrays. This carries the advantage of using existing JAVA and JML features and thus making the specification also readable for other JML tools. The class `MemoryStack` could be declared and specified in the following way: A so-called JML ghost field[3] `_stack` stores the internal state of a MemoryStack. The `push` method returns a new `MemoryStack` with a new internal array `_stack` that contains all elements from the original stack plus the one pushed on it. A possible specification for `MemoryStack`, containing in addition to `push` several auxiliary methods, is given by:

— Java + JML —

```
public class MemoryStack{

   //@ public invariant \inImmortalMemory(this);

   //@ private ghost MemoryArea[] _stack;

   //@ public static invariant EMPTY_STACK.size()==0;
   private static MemoryStack EMPTY_STACK;


   /*@ public invariant
     @    (\forall int i,j;
     @     0<=i && i<_stack.length && i<j && j<_stack.length;
     @     _stack[i]!=null && _stack[i]!=_stack[j]);
     @*/

   /*@ public normal_behavior
     @  working_space 0;
     @  ensures \fresh(\result) && \fresh(\result._stack) &&
     @     \result._stack.length==_stack.length+1 &&
     @     \result._stack[_stack.length]==m &&
     @     (\forall int i; i>=0 && i<_stack.length;
     @                    \result._stack[i]==_stack[i]);
     @*/
   public MemoryStack push(MemoryArea m);

   /*@ public normal_behavior
     @  ensures \result==(\exists int i; i>=0 && i<_stack.length;
     @                                 _stack[i]==m);
     @*/
   public /*@pure@*/ boolean contains(MemoryArea m);

   /*@ public normal_behavior
     @  ensures \result==((o instanceof MemoryStack) &&
     @      _stack.length == ((MemoryStack) o)._stack.length &&
     @      (\forall int i; 0<=i && i<_stack.length;
     @       _stack[i]==((MemoryStack) o)._stack[i]));
```

[3] A ghost field is a field declared with the JML modifier **ghost** within a JML specification. It is only available for specification purposes and not visible on the implementation level.

```
      @*/
    public /*@pure@*/ boolean equals(Object o);

    /*@ public normal_behavior
      @  ensures \result==_stack[i];
      @*/
    public /*@pure@*/ MemoryArea get(int i);

    /*@ public normal_behavior
      @  ensures \result==_stack.length;
      @*/
    public /*@pure@*/ int size(){return _stack.length;}

    /*@ public normal_behavior
      @  ensures \result==_stack[_stack.length-1];
      @*/
    public /*@pure@*/ MemoryArea top();
}
```

Java + JML

The above specification incorporates a newly defined JML construct **\inImmortalMemory** which indicates that its argument is allocated in immortal memory. Since every scope stack is allocated in immortal memory, it can legally be referenced from any scope. The drawback of this specification is that it is more involved than needed leading to bigger and harder to handle proofs.

The following considerations lead us to a simpler specification: On no occasions it is necessary to know the exact structure of the scope stack. For checking the single parent rule it is sufficient for each scope to know its parent scope which could also be stored by an attribute. For determining whether an assignment operation is legal or not we only need to know if one scope is an inner scope of another scope which can be described by a binary relation.

The Sub Stack Relation $\preceq$

A more lightweight way of formalizing the behavior of the scope stack than the previously presented is to define a sub stack relation $\preceq$, where $a \preceq b$ is true for two local stacks a and b if a is a prefix of b. The formal specification of push then just has to express that the result is a newly created MemoryStack and that

$$\texttt{s} \preceq \texttt{s.push(a)}$$

holds for every stack s and every scope a which could be expressed as a JML method specification as shown in Figure 6.2. In Figure 6.2 we denote $\preceq$ as **\subStack**. Due to the immutability of MemoryStack, $\preceq$ can be rigid (its arguments, however, can be flexible) which makes it less involved to handle in dynamic logic than it would be the case for a flexible symbol. However, the **stack** attribute of a scope can be set to a different value during the program run. This means that each instance of MemoryStack is only a snapshot of a part of the scope stack taken at a certain time.

— Java + JML —

```
/*@ public normal_behavior
  @  ...
  @  ensures \fresh(\result) &&
  @       \subStack(this, \result);
  @*/
public MemoryStack push(ScopedMemory m);
```

— Java + JML —

Figure 6.2: JML specification of `push`

Definition 6.1 (Syntax and Semantics of Relation $\preceq$) *$\preceq$ is a binary predicate with*

$$\alpha(\preceq) := MemoryStack \times MemoryStack$$

Let s be a state, β a variable assignment and a and b two terms with $a, b \in Terms_{MemoryStack}$ then a represents a sub stack of b in state s if and only if

$$s, \beta \models a \preceq b$$

In addition for each state s, variable assignment β and term a with $a \in Terms_{MemoryStack}$:

$$\begin{array}{rcl} s, \beta & \not\models & a \preceq \textbf{\texttt{null}} \\ s, \beta & \not\models & \textbf{\texttt{null}} \preceq a \end{array}$$

From definition 6.1 follows that given a state s, a variable assignment β and two terms a and b with $\delta(val_{s,\beta}(a)) \sqsubseteq \texttt{MemoryArea}$ and $\delta(val_{s,\beta}(b)) \sqsubseteq \texttt{MemoryArea}$, a represents an outer scope of b in state s if and only if

$$s, \beta \models a.\texttt{stack} \preceq b.\texttt{stack}$$

As described in Section 2.3.3, Java DL operates under the constant domain semantics which means all instances that can ever be created by the program have to exist from the beginning. Since $\preceq$ is rigid, the createdness of objects does not influence its evaluation and it thus has also to be defined on not yet created objects. To make sure that we can always create a stack of which the current stack is a substack, we have to require that for every integer value i there are infinitely many integer values j such that

$$j > i \text{ and } I(get_{MemoryStack})(i) \preceq^{I} I(get_{MemoryStack})(j)$$

holds for the interpretations I of every Kripke seed $(\mathcal{D}, I)$. This is, however, already ensured as a byproduct of the specification of `push` which states that there is a fresh instance of `MemoryStack` which meets this requirement.

For efficiency reasons, we encode this behavior of the scope stack in the newly defined rule *push*:

$$\mathsf{push}\ \frac{\begin{array}{l}\Gamma,\ \{\mathcal{U}\}(b \preceq get_{MemoryStack}(j) \wedge j \geq MemoryStack.\texttt{<ntc>}) \Rightarrow \\ \quad \{\mathcal{U}\}\{\ a := get_{MemoryStack}(j)\ || \\ \qquad MemoryStack.\texttt{<ntc>} := j+1\ || \\ \qquad for\ i;\ if(MemoryStack.\texttt{<ntc>} \leq i \wedge i \leq j) \\ \qquad\qquad get_{MemoryStack}(i).\texttt{<c>} := TRUE\ \ || \\ \qquad get_{MemoryStack}(j).\,\texttt{<ma>} := b.\,\texttt{<ma>}\}\langle \pi\, \omega\rangle\, \phi,\ \Delta \end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle \pi\ \texttt{a=b.push(s)@(MemoryStack);}\ \omega\rangle\, \phi,\ \Delta}$$

where j is a fresh integer skolem constant. By adding the assumption that

$$j \geq MemoryStack.\texttt{<ntc>}$$

and setting a to $get_{MemoryStack}(j)$, we can express that the result of `push` is some newly created `MemoryStack`. Note, that we cannot just simply set b to

$$get_{MemoryStack}(MemoryStack.\texttt{<ntc>})$$

since then we could deduce from this rule by induction that

$$\begin{aligned} \forall\, int\; x, y;\;\; & (x \geq MemoryStack.\texttt{<ntc>} \land y \geq x) \rightarrow \\ & get_{MemoryStack}(x) \preceq get_{MemoryStack}(y) \end{aligned}$$

which is obviously false.

Having defined the calculus rule *push* the class `MemoryStack` can be reduced to:

— Java + JML —

```
public class MemoryStack{

    //@ public invariant \inImmortalMemory(this);

    public MemoryStack push(MemoryArea m);

}
```

— Java + JML —

This specification still states that all instances of `MemoryStack` are allocated in immortal memory. As JML invariants only refer to already created objects, we need the update

$$\{get_{MemoryStack}(j).\texttt{<ma>} := b.\texttt{<ma>}\}$$

in rule *push* to preserve this invariant.

In summary, describing the scope stack by the relation $\preceq$ is more suitable for our purposes than the alternative explicit modeling based on arrays. In the remainder of this chapter we will therefore base our considerations only on the specification of `MemoryStack` using $\preceq$.

The Predicate im

For distinguishing the scope stack of the immortal memory area from other scope stacks we introduce the predicate im with $\mathsf{im}(s)$ evaluating to *true* if and only if s is the scope stack of the immortal memory area. For the JML construct **`\inImmortalMemory`** we consequently define:

$$T^{DL}_{JML}(\backslash\texttt{inImmortalMemory(o)}) := \mathsf{im}(o.\texttt{<ma>}.\mathtt{stack})$$

6.3.2 Specification of `MemoryArea` and its Sub-Types

The behaviour of memory areas is described by an implementation of class `MemoryArea` and its subtypes augmented with class invariants expressed in JML. Due to the restrictions (see Section 5) imposed on the memory model, all memory areas we permitted (immortal and scoped memory) behave basically like scoped memory areas: since calling the `enter` method on the immortal memory area is no longer allowed its behavior is basically identical to that of a scoped memory area. The mere difference of the immortal memory compared to "normal" scopes is its distinct position at the bottom of the scope stack. Due to this uniformity, it is sufficient to provide a reference implementation for `javax.realtime.MemoryArea`: the classes `ScopedMemory`, `LTMemory` and `VTMemory` do not need to override any of the `enter` and `executeInArea` methods since their behaviour is sufficiently described by the implementation and specification of `MemoryArea`. Only for `ImmortalMemory` we have to strengthen (compared to `MemoryArea`) its class invariants a bit for the afore mentioned reasons.

Attributes and Invariants

Each memory area possesses its own scope stack accessible via the attribute **`stack`**. Due to the chosen lightweight modeling of the scope stack, we need an additional attribute `parent` for determining the parent scope.

— Java + JML —

```
public abstract class MemoryArea{

    public MemoryStack /*@nullable@*/ stack;
    public /*@nullable@*/ MemoryArea parent;
    protected int referenceCount=0;
    protected final long size;
    protected long consumed;
    protected final /*@nullable@*/ java.lang.Runnable logic;

    /*@ public invariant referenceCount>=0 &&
      @     (referenceCount>0 <==> parent!=null); @*/
    //@ public invariant parent==null <==> stack==null;
    //@ public invariant parent==null ==> consumed==0;

    ...
```

— Java + JML —

The attributes `size` and `consumed` determine the size of the scope in bytes and how much of it has already been consumed. The number of executions of the scopes `enter` method currently in progress is counted by `referenceCount`. The attribute `logic` stores the `Runnable` object passed to this memory area by its constructor. Whenever the parameterless version of the `enter()` method is called, the `run()` method of `logic` is executed.

For the `enter` method, we provide a reference implementation that

- ensures that the `Runnable` passed to the `MemoryArea` is not **`null`** and that the single parent rule is met.

- If that is the case parent is set to <cma> and referenceCount is increased
- If the memory area is not yet on the scope stack, the attribute **stack** is initialised by a new stack created by <cma>.stack.push(**this**).
- Afterwards the implicit method <runRunnable> is called, which switches cma to **this** and executes logic.
- If the execution of logic raises an exception we have to check whether the exception was raised in this memory area. If this is the case a ThrowBoundaryError is raised. For this purpose we use the ThrowBoundaryError given by RealtimeSystem.TBE which is pre-allocated in immortal memory. The rationale behind using pre-allocated exceptions and defining a class RealtimeSystem which provides them is explained in Section 6.3.4.
- Before leaving the enter method, the reference count is decreased again. If it is equal to 0, indicating that no thread executes in the considered memory area anymore, parent and stack are reset to **null** and the amount of consumed memory in the area to 0. The implicit method <delete> deletes all objects in this scope by moving them to scope RealtimeSystem.TRASH (see Section 6.3.4). The exact semantics of <delete> is defined in Section 6.5. We do not model finalization of objects in freed memory areas. Note, that these objects are not accessible in the remainder (after the memory is freed) of the program, therefore, and due to the constraints on finalization imposed in Section 5.2, treating deletion of objects in this way does not influence the modeled semantics of RTSJ programs.

Since we do not allow Java threads but only RTSJ Schedulable objects in KeYSCJ, we do not need to take into account java.lang.IllegalThreadStateExceptions arising from methods of MemoryArea being called by Java threads.

— Java —

```
public abstract class MemoryArea{
    ...

    public void enter(java.lang.Runnable logic){
        if(logic==null) throw new IllegalArgumentException();
        if(parent!=null && parent!=<cma>){
            throw new ScopedCycleException();
        }
        parent = <cma>;
        referenceCount++;
        if(stack==null){
            stack = <cma>.stack.push(this);
        }
        try{
            <runRunnable>(logic);
        }catch(Exception e){
            if(this==getMemoryArea(e)){
                throw RealtimeSystem.TBE;
            }
        }finally{
            referenceCount--;
            if(referenceCount==0){
                <delete>(RealtimeSystem.TRASH);
                consumed=0;
                parent=null;
                stack=null;
            }
        }
    }

    ...
}
```

Java

The `executeInArea` method is implemented in a similar manner. Instead of checking the single parent rule, which is not necessary since the stack is not changed by `executeInArea`, we have to ensure that the considered memory area is accessible, i.e., that it is located below `<cma>` on the scope stack:

Java

```
public abstract class MemoryArea{
    ...

    public void executeInArea(java.lang.Runnable logic){
        if(logic==null) throw new IllegalArgumentException();
        if(!outerScopeM(this, <cma>)){
          throw new InaccessibleAreaException();
        }
        try{
          <runRunnable>(logic);
        }catch(Exception e){
            if(this==getMemoryArea(e)){
              throw RealtimeSystem.TBE;
            }
        }
    }

    ...
}
```

Java

In the above reference implementations we made use of the helper method `outerScopeM` and the RTSJ API method `getMemoryArea` for which we provided the JML specification shown below [4]. The implementation of the implicit method `<runRunnable>` is rather simple; it just

[4] The specification of `outerScopeM` makes use of the construct `\outerScope(a,b)` (expressing that `a` is an outer scope of `b`). Its mapping to Java DL is defined in Section 7.

calls the `run` method of the `Runnable` passed to it. The important part of `<runRunnable>`'s functionality namely the switching of the currently active memory area is performed by rule expandRR presented in Section 6.2.1.

Java + JML

```
public abstract class MemoryArea{
    ...

    private void <runRunnable>(java.lang.Runnable logic){
        logic.run();
    }

    private void <delete>(MemoryArea m);

    /*@ public normal_behavior
      @  working_space 0;
      @  ensures \result==\outerScope(a,b);
      @*/
    public static /*@pure@*/ boolean outerScopeM(MemoryArea a,
                                                  MemoryArea b);

    /*@ public normal_behavior
      @  ensures \result == \memoryArea(object);
      @  working_space 0;
      @*/
    public static /*@pure@*/ MemoryArea getMemoryArea(java.lang.Object object);

    ...
}
```

Java + JML

As explained before, the immortal memory area behaves like a scoped memory area of which we know that

- it is a singleton, and
- it is located at the bottom of the scope stack.

The following specification expresses that immortal memory areas can only occur as a singleton and that the immortal memory area itself is allocated in immortal memory. All other relevant properties of `ImmortalMemory`, such as being an outer scope for every scope on the stack, are expressed by the calculus rules shown in Section 6.4.

Java + JML

```
public class ImmortalMemory extends MemoryArea{

    /*@ public static invariant
      @         (\forall ImmortalMemory i; i==instance) &&
      @         \memoryArea(instance)==instance &&
      @        instance.parent==instance &&
      @        \inImmortalMemory(instance);
      @*/
    private static ImmortalMemory instance;

    /*@ public normal_behavior
      @  assignable \nothing;
      @  ensures \result == instance;
      @*/
    public static /*@pure@*/ ImmortalMemory instance(){
        return instance;
    }
}
```

Java + JML

The behaviour of instances of `ScopedMemory` is almost sufficiently described by the reference implementation of `MemoryArea`. One particularity of scoped memory areas that remains uncovered by this implementation are *portals*. This aspect is taken care of by the implementation of `ScopedMemory`.

Java + JML

```
public abstract class ScopedMemory extends MemoryArea{

  private /*@nullable@*/ Object portal;
  ...
  public java.lang.Object getPortal(){
    if(outerScopeM(this, <cma>)){
      return portal;
    }else{
      throw new IllegalAssignmentError();
    }
  }

  public void setPortal(java.lang.Object object){
    if(!outerScopeM(this, <cma>)){
      throw new InaccessibleAreaException();
    }
    if(object!=null){
      if(this==getMemoryArea(object)){
        portal = object;
      }else{
        throw new IllegalAssignmentError();
      }
    }
  }
}
```

Java + JML

A portal can only be obtained from a scope a via the `getPortal()` if for the callers active scope b

$$a.\texttt{stack} \preceq b.\texttt{stack}$$

holds. If this is not the case an `IllegalAssignmentError` is raised. This condition has also to be met in case a portal is set via the `setPortal` method leading to an

`InaccessibleAreaException`

otherwise. Portal objects must be allocated in the scoped memory area to which they serve as a gateway. Calling `setPortal(o)` leaves the existing portal unchanged if `o` equals **`null`** and raises an `IllegalAssignmentError` if and only if `o` is not allocated in this memory scope.

6.3.3 Admissible References to Inner Scopes

The RTSJ constrains non-static references to point only to the same or to an outer scope. In our model, there are, however, a few exceptions to this: (i) the implicit field <ma>`@Object` and (ii) the field `parent@MemoryArea`.

The reason for the first exception is, that due to the `executeInArea` method we cannot assume that the scope `s1` in which the object representing a scope `s2` is allocated in is an outer scope of `s2` for every possible stack that can be built by an application. The RTSJ code from Figure 6.3 creates such a stack (shown in Figure 6.4) in which the scope `ltm3` is not allocated in one of its outer scopes. For an object `o` allocated in `ltm3` the reference `o.`<ma> is not a legal RTSJ reference since it points to the object `ltm3` which is not allocated in `ltm3` or one of its enclosing scopes.

The second exception concerning the attribute `parent@MemoryArea` has to be made since scopes can be entered in a different order than they are created as shown in Figure 6.5: The object `s3` is allocated in a more outer scope than `s4` (`s3.`<ma>`.stack` $\preceq$ `s4.`<ma>`.stack`) but still the scope `s3` is an inner scope of `s4` (`s4.stack` $\preceq$ `s3.stack`).

6.3.4 The Helper Class `RealtimeSystem`

We model certain exceptions, namely `OutOfMemoryError` (which will become important in Section 10) and `ThrowBoundaryError`, that can be thrown by KeYSCJ programs as being preallocated in immortal memory. For obtaining these exceptions we define a class `RealtimeSystem` which is shown in figure 6.6. The reason for modeling these exceptions as preallocated is twofold. Modeling object creation introduces a certain overhead to automatic proof finding thus using preallocated exceptions instead of dynamically creating increases efficiency. In addition especially `ThrowBoundaryErrors` and `OutOfMemoryErrors` can entail a cascade of additional exceptions if they are dynamically created in scoped memory:

- An `OutOfMemoryError` is thrown if an object allocation cannot be performed due to little remaining free memory in a scope. The dynamic creation of an `OutOfMemoryError` would then just trigger a new `OutOfMemoryError`.

- A `ThrowBoundaryError` is thrown if another thrown exception is not caught before reaching the scope *boundary*. Throwing a `ThrowBoundaryError` that was allocated in scoped memory would cause a new `ThrowBoundaryError` as soon as this error reaches another scope boundary.

Java + JML

```
final LTMemory ltm1 = new LTMemory(3000000);
final Runnable hello = new Runnable(){...};
ltm1.enter(new Runnable(){
  public void run(){
    final LTMemory ltm2 = new LTMemory(3000000);
    ltm2.enter(new Runnable(){
      public void run(){
        final LTMemory ltm3 = new LTMemory(3000000);
        ltm1.executeInArea( new Runnable(){
          public void run(){
            ltm3.enter(hello);
          }
        });
      }
    });
  }
});
```

Java + JML

Figure 6.3: A piece of code building a branched cactus stack

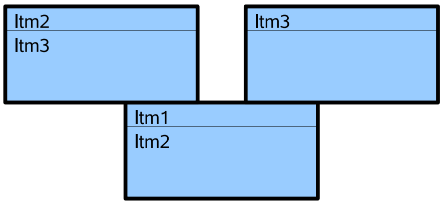


Figure 6.4: Scope stack created by the code from Figure 6.3

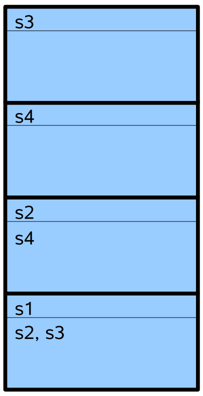


Figure 6.5: `s3` is entered from `s4` which is allocated in a more inner scope than `s3`

— Java + JML —

```
package javax.realtime;

public class RealtimeSystem{

  //@ public static invariant \inImmortalMemory(OOME);
  //@ public static invariant \inImmortalMemory(TBE);
  //@ public static invariant \inImmortalMemory(TRASH);

  public static final OutOfMemoryError OOME =
      new OutOfMemoryError();

  public static final ThrowBoundaryError TBE =
      new ThrowBoundaryError();

  public static final ScopedMemory TRASH =
      new LTMemory();

}
```

— Java + JML —

Figure 6.6: The class `RealtimeSystem` providing preallocated exceptions

By avoiding these consecutive exceptions failed proofs are more simple to analyze which is important in interactive proving for finding the cause of the failure needed to adapt the code and specification that is to be verified accordingly.

The use of pre-allocated exceptions is permitted by the RTSJ. There are also implementations (such as [Dawson, 2008]) of the RTSJ following this approach.

In addition the class `RealtimeSystem` also provides a reference (`TRASH`) to the scope holding deleted objects.

6.4 Axiomatization of $\preceq$ and im

By what has been described so far $\preceq$ is merely an uninterpreted predicate, so we have to axiomatize what it should semantically stand for. This is done via calculus rules as demonstrated in the following. The relation $\preceq$ is a partial order and thus reflexive, transitive and antisymmetric:

$$\mathsf{outerScopeReflexive}\ \frac{\Gamma \Rightarrow \mathit{true},\ o \doteq \mathtt{null},\ \Delta}{\Gamma \Rightarrow o \preceq o,\ o \doteq \mathtt{null},\ \Delta}$$

$$\mathsf{outerScopeTransitive}\ \frac{\Gamma,\ o_1 \preceq o_2,\ o_2 \preceq o_3,\ o_1 \preceq o_3 \Rightarrow \Delta}{\Gamma,\ o_1 \preceq o_2,\ o_2 \preceq o_3 \Rightarrow \Delta}$$

$$\mathsf{outerScopeAntisymmetric}\ \frac{\Gamma,\ o_1 \preceq o_2,\ o_2 \preceq o_1,\ o_1 \doteq o_2 \Rightarrow \Delta}{\Gamma,\ o_1 \preceq o_2,\ o_2 \preceq o_1 \Rightarrow \Delta}$$

Also directly from definition 6.1 follows that for two arbitrary terms a and b with $a, b \in Terms_{MemoryStack}$

$$a \preceq b \rightarrow a \not\doteq \texttt{\textbf{null}} \land b \not\doteq \texttt{\textbf{null}}$$

is valid. This is expressed by rule nonNullStack:

$$\textsf{nonNullStack}\ \frac{\Gamma,\ a \preceq b \Rightarrow a \doteq \texttt{\textbf{null}},\ b \doteq \texttt{\textbf{null}},\ \Delta}{\Gamma,\ a \preceq b \Rightarrow \Delta}$$

Since the attribute `stack` is either set to **`null`** or to some newly created object during program execution, two different scopes cannot share the same local stack (unequal **`null`**). This means the axiom

$$\forall\ MemoryArea\ a, b; \begin{pmatrix} a.{<}\texttt{c}{>} \doteq \mathit{TRUE} \land \\ b.{<}\texttt{c}{>} \doteq \mathit{TRUE} \land \\ a.\texttt{stack} \not\doteq \texttt{\textbf{null}} \land \\ a.\texttt{stack} \doteq b.\texttt{stack} \end{pmatrix} \rightarrow a \doteq b \tag{6.2}$$

holds in all *reachable* states.

We make use of this axiom by rule stackInjective:

$$\textsf{stackInjective}\ \frac{\Gamma,\ a.{<}\texttt{c}{>} \doteq \mathit{TRUE},\ b.{<}\texttt{c}{>} \doteq \mathit{TRUE},\ a \doteq b,\ \mathrm{RS} \Rightarrow a.\texttt{stack} \not\doteq \texttt{\textbf{null}},\ \Delta}{\Gamma,\ a.{<}\texttt{c}{>} \doteq \mathit{TRUE},\ b.{<}\texttt{c}{>} \doteq \mathit{TRUE},\ a.\texttt{stack} \doteq b.\texttt{stack},\ \mathrm{RS} \Rightarrow a.\texttt{stack} \not\doteq \texttt{\textbf{null}},\ \Delta}$$

The following rule states that in every reachable program state all non-static attributes different from `<ma>@Object` and `parent@MemoryArea`, that are not **`null`**, point to the same or an outer scope:

$$\textsf{outerRefAttr1}\ \frac{\begin{array}{l}\Gamma,\ o.{<}\texttt{c}{>} \doteq \mathit{TRUE},\ o.a.\texttt{<ma>.stack} \preceq o.\texttt{<ma>.stack},\ \mathrm{RS}\ \Rightarrow \\ o.a \doteq \texttt{\textbf{null}},\ \Delta\end{array}}{\Gamma,\ o.{<}\texttt{c}{>} \doteq \mathit{TRUE},\ \mathrm{RS} \Rightarrow o.a \doteq \texttt{\textbf{null}},\ \Delta}$$

Analogously objects referenced by array slots are allocated in an outer or the same scope compared to the scope containing the array itself:

$$\textsf{outerRefArray1}\ \frac{\begin{array}{l}\Gamma,\ arr.{<}\texttt{c}{>} \doteq \mathit{TRUE},\ arr[i].\texttt{<ma>.stack} \preceq arr.\texttt{<ma>.stack},\ \mathrm{RS}\ \Rightarrow \\ arr[i] \doteq \texttt{\textbf{null}},\ \Delta\end{array}}{\Gamma,\ arr.{<}\texttt{c}{>} \doteq \mathit{TRUE},\ \mathrm{RS} \Rightarrow arr[i] \doteq \texttt{\textbf{null}},\ \Delta}$$

This is a redundant rule defined to facilitate shorter proofs and increase automation:

$$\textsf{outerRefAttr2}\ \frac{\begin{array}{l}\Gamma,\ o.{<}\texttt{c}{>} \doteq \mathit{TRUE},\ \mathrm{RS} \Rightarrow \\ o.a \doteq \texttt{\textbf{null}},\ o.a.\texttt{<ma>.stack} \preceq s, o.\texttt{<ma>.stack} \preceq s,\ \Delta\end{array}}{\begin{array}{l}\Gamma,\ o.{<}\texttt{c}{>} \doteq \mathit{TRUE},\ \mathrm{RS} \Rightarrow \\ o.a \doteq \texttt{\textbf{null}},\ o.a.\texttt{<ma>.stack} \preceq s,\ \Delta\end{array}}$$

Given the above conclusion the premise can as well be derived from outerScopeTransitive, outerRef1 and the cut rule:

$$\frac{\dfrac{\dfrac{\dfrac{*}{\Gamma,\ \Gamma''',\ o.a.\texttt{<ma>.stack} \preceq s,\ \mathrm{RS} \Rightarrow \Delta',\ \Delta}}{\Gamma,\ \Gamma'',\ o.\texttt{<ma>.stack} \preceq s,\ \mathrm{RS} \Rightarrow \Delta',\ \Delta} \quad \Gamma,\ \Gamma'',\ \mathrm{RS} \Rightarrow o.\texttt{<ma>.stack} \preceq s, \Delta',\ \Delta}{\Gamma, \Gamma',\ o.a.\texttt{<ma>.stack} \preceq o.\texttt{<ma>.stack},\ \mathrm{RS} \Rightarrow \Delta',\ \Delta}}{\Gamma,\ o.{<}\texttt{c}{>} \doteq \mathit{TRUE},\ \mathrm{RS} \Rightarrow o.a \doteq \texttt{\textbf{null}},\ o.a.\texttt{<ma>.stack} \preceq s,\ \Delta}$$

where

- $\Delta' := \{o.a \doteq \texttt{null},\ o.a.\texttt{<ma>}.\texttt{stack} \preceq s\}$,
- $\Gamma' := \{o.\texttt{<c>} \doteq \mathit{TRUE}\}$,
- $\Gamma'' := \Gamma' \cup \{o.a.\texttt{<ma>}.\texttt{stack} \preceq o.\texttt{<ma>}.\texttt{stack}\}$ and
- $\Gamma''' := \Gamma'' \cup \{o.\texttt{<ma>}.\texttt{stack} \preceq s\}$.

In correspondence to outerRefAttr2 we can define outerRefArray2:

$$\textsf{outerRefArray2}\ \frac{\begin{array}{l}\Gamma,\ arr.\texttt{<c>} \doteq \mathit{TRUE},\ \mathrm{RS} \Rightarrow \\ arr[i] \doteq \texttt{null},\ arr[i].\texttt{<ma>}.\texttt{stack} \preceq s, arr.\texttt{<ma>}.\texttt{stack} \preceq s,\ \Delta\end{array}}{\begin{array}{l}\Gamma,\ arr.\texttt{<c>} \doteq \mathit{TRUE},\ \mathrm{RS} \Rightarrow \\ arr[i] \doteq \texttt{null},\ arr[i].\texttt{<ma>}.\texttt{stack} \preceq s,\ \Delta\end{array}}$$

Memory areas are represented by JAVA objects which are themselves allocated in a certain memory area. This imposes a hierarchy $\preceq'$ on the set of memory areas of an application that is different from the cactus stack order $\preceq$.

Definition 6.2 *Let M be a set of scopes with $M \subseteq \mathbf{U}$, then for every state s and variable assignment β the relation $s(\preceq') \subseteq M^2$ is minimal such that :*

$$s, \beta \models a \preceq' b \qquad \mathit{if} \quad s, \beta \models b.\texttt{<ma>} \doteq a \lor \exists\ \mathit{MemoryArea}\ c;\ (b.\texttt{<ma>} \doteq c \land a \preceq' c)$$

The substack relation $\preceq$ has to respect $\preceq'$ to that extend that no scope can be allocated in one of its inner scopes or, in other words, for each two memory areas a, b occurring on the cactus stack

$$a.\texttt{<ma>} \preceq' b.\texttt{<ma>} \rightarrow \neg b.\texttt{<ma>}.\texttt{stack} \preceq a.\texttt{<ma>}.\texttt{stack} \tag{6.3}$$

holds in every *reachable* state. This entails that

$$\mathrm{RS} \rightarrow \forall\ \mathit{MemoryArea}\ m; (m.\texttt{<c>} \doteq \mathit{TRUE} \rightarrow \neg m.\texttt{stack} \preceq m.\texttt{<ma>}.\texttt{stack}) \tag{6.4}$$

is valid which leads to the rewrite rule:

$$\textsf{maAllocOuter}\ \frac{[m.\texttt{stack} \preceq m.\texttt{<ma>}.\texttt{stack} \rightsquigarrow \mathit{false}]}{\Gamma,\ m.\texttt{<c>} \doteq \mathit{TRUE},\ \mathrm{RS} \Rightarrow \Delta}$$

where m is a term of type `MemoryArea`. As Figure 6.4 illustrates, we can, however, not deduce from this that $a.\texttt{<ma>} \preceq' b.\texttt{<ma>} \rightarrow a.\texttt{<ma>}.\texttt{stack} \preceq b.\texttt{<ma>}.\texttt{stack}$ since $\preceq$ is not total.

Immortal memory always occurs as a singleton. Since the stack associated with this immortal scope does not change either during the program run, we can assume that there can be only one stack for which im holds:

$$\textsf{imUnique}\ \frac{\Gamma,\ \mathsf{im}(o_1),\ \mathsf{im}(o_2),\ o_1 \doteq o_2 \Rightarrow \Delta}{\Gamma,\ \mathsf{im}(o_1),\ \mathsf{im}(o_2) \Rightarrow \Delta}$$

The immortal scope is the outermost scope on the scope stack. Thus the only stack that is a sub stack of the stack belonging to the immortal scope is the stack of the immortal scope itself:

$$\textsf{imSub1}\ \frac{\Gamma,\ o_1 \preceq o_2,\ \mathsf{im}(o_2),\ o_1 \doteq o_2 \Rightarrow \Delta}{\Gamma,\ o_1 \preceq o_2,\ \mathsf{im}(o_2) \Rightarrow \Delta}$$

The stack of the immortal scope is a sub stack of any other stack. This is expressed by the rewrite rule:

$$\mathsf{imSub2}\ \frac{[o_1 \preceq o_2 \rightsquigarrow \mathit{true}]}{\Gamma,\ \mathsf{im}(o_1) \Rightarrow \Delta}$$

For any object `o` and array `arr` (with reference type elements) allocated in immortal memory the objects referenced by `o.a` (for an attribute `a` having a reference type) and `arr[i]` must be allocated in immortal memory as well. Conversely, this means:

> If `o.a` (`arr[i]`) is not allocated in immortal memory than `o` (`arr`) cannot be either.

This can already be derived from the rules we have defined so far but would require using the cut rule which can be obstructive for automated proof finding. Therefore, we introduce the following two rules:

$$\mathsf{imSub3}\ \frac{\begin{array}{c}\Gamma,\ o.\texttt{<c>} \doteq \mathit{TRUE},\ \mathrm{RS} \Rightarrow \\ \mathsf{im}(o.\texttt{<ma>}\texttt{.stack}),\ \mathsf{im}(o.a.\texttt{<ma>}\texttt{.stack}),\ o.a \doteq \texttt{null},\ \Delta\end{array}}{\Gamma,\ o.\texttt{<c>} \doteq \mathit{TRUE},\ \mathrm{RS} \Rightarrow \mathsf{im}(o.a.\texttt{<ma>}\texttt{.stack}),\ o.a \doteq \texttt{null},\ \Delta}$$

$$\mathsf{imSub4}\ \frac{\begin{array}{c}\Gamma,\ \mathit{arr}.\texttt{<c>} \doteq \mathit{TRUE},\ \mathrm{RS} \Rightarrow \\ \mathsf{im}(\mathit{arr}.\texttt{<ma>}\texttt{.stack}),\ \mathsf{im}(\mathit{arr}[i].\texttt{<ma>}\texttt{.stack}),\ o.a \doteq \texttt{null},\ \Delta\end{array}}{\Gamma,\ \mathit{arr}.\texttt{<c>} \doteq \mathit{TRUE},\ \mathrm{RS} \Rightarrow \mathsf{im}(\mathit{arr}[i].\texttt{<ma>}\texttt{.stack}),\ \mathit{arr}[i] \doteq \texttt{null},\ \Delta}$$

where $arr[i]$ is a reference type array slot and a a reference type attribute Finally, provided that we abstract away from static initialisation, we can assume that a reference type static field is either **`null`** or created and referencing an object allocated in immortal memory:

$$\mathsf{staticIm}\ \frac{\Gamma,\ \mathit{sa}.\texttt{<c>} \doteq \mathit{TRUE},\ \mathsf{im}(\mathit{sa}.\texttt{<ma>}\texttt{.stack}),\ \mathrm{RS} \Rightarrow \mathit{sa} \doteq \texttt{null},\ \Delta}{\Gamma, \mathrm{RS} \Rightarrow \mathit{sa} \doteq \texttt{null},\ \Delta}$$

where sa is a reference type static field.

6.5 The Implicit Method `<delete>`

For treating RTSJ programs we have to take into account deletion of objects. The alternative (i.e., just leaving deleted objects in their original memory area and not distinguishing them from non-deleted objects) could render references pointing from or to deleted objects illegal (due to a changing of the local scope stack and the nesting hierarchy when a scope is popped from the cactus stack and later pushed on it again) making some of the axioms assumed in Section 6.4 invalid and the corresponding taclets thus unsound.

The basic idea behind the modeling of object deletion is as follows: All objects allocated in a scope s that is to be freed are moved to the scope `RealtimeSystem.TRASH` dedicated to holding deleted objects. The local stack of `RealtimeSystem.TRASH` is changed in that way, that all legal references pointing from or to objects in s or `RealtimeSystem.TRASH` remain legal after the deletion is performed. The described procedure is performed by the implicit method `<delete>` whose semantics is defined by taclet deleteScope:

deleteScope

$$\frac{\begin{array}{l}
\Gamma,\ \{\mathcal{U}\}\left(\begin{array}{l}
trash.stack \preceq get_{MemoryStack}(j)\wedge\\
se.stack \preceq get_{MemoryStack}(j)\wedge\\
\forall\, MemoryArea\, m; \left(\begin{array}{l}\left(\begin{array}{l} m.created \doteq TRUE \wedge\\ m \not\doteq trash \wedge m \not\doteq se\wedge\\ \left(\begin{array}{l} se.stack \preceq m.stack\vee\\ trash.stack \preceq m.stack\end{array}\right)\end{array}\right) \rightarrow\\ get_{MemoryStack}(j) \preceq m.stack\end{array}\right)\wedge\\
j \geq MemoryStack.\texttt{<ntc>}
\end{array}\right)\\
\Rightarrow\\
\{\mathcal{U}\}\{\, trash.\texttt{stack} := get_{MemoryStack}(j) \mid\mid\\
\qquad MemoryStack.\texttt{<ntc>} := j+1 \mid\mid\\
\qquad for\ i;\ if(MemoryStack.\texttt{<ntc>} \leq i \wedge i \leq j)\\
\qquad\qquad\qquad\qquad get_{MemoryStack}(i).\texttt{<c>} := TRUE\ \mid\mid\\
\qquad get_{MemoryStack}(j).\,\texttt{<ma>} := trash.\,\texttt{<ma>} \mid\mid\\
\qquad for\ Object\ o;\ if(o.\,\texttt{<ma>} \doteq se)o.\,\texttt{<ma>} := trash\}\langle\pi\,\omega\rangle\,\phi,\ \Delta
\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle\pi\ \texttt{se.<delete>(trash)@(MemoryArea);}\ \omega\rangle\,\phi,\ \Delta}$$

where j is a fresh integer skolem constant. The update in the succedent sets $trash.\texttt{stack}$ to a newly created `MemoryStack` and moves all objects from scope *se* to *trash*. In the antecedent we formulate constraints for the new stack, namely that

- the stacks of *trash* and *se* are substacks of it and
- all scopes that were inner scopes of *trash* or *se* are inner scopes of *trash* after resetting the stack of *trash*. Thus, also all references from objects allocated in these scopes and pointing to objects allocated in *trash* or *se* remain legal[5].

6.6 Proof Obligation for RS

In the previous Section we defined several novel axioms which we required to hold in *reachable states*. As pointed out in Section 2.3.4, the necessity may occur to prove that RS holds in a certain state which, from a technical point of view, happens when we encounter sequents of the form:

$$\Gamma,\ \mathrm{RS} \Rightarrow \{\mathcal{U}\}\,\mathrm{RS},\ \Delta$$

As described in Section 2.3.4, this is done by replacing the occurrence of RS in the scope of update $\mathcal{U}$ with a conjunction Φ of those axioms (characterizing RS) that can possibly be affected by the update $\mathcal{U}$.

For the newly defined axioms from the previous Section we have to proceed analogously. Concretely, this means that we have to ensure that

[5] Note, that for any legal execution trace of an RTSJ program there can be, of course, no inner scopes of a scope whose memory is reclaimed. However, due to our approach underspecifying the structure of the scope stack, we need to add this constraint. Otherwise it might not be possible to deduce that the RS predicate is preserved (see Section 6.6) in certain cases even though it actually should be according to the semantics of RTSJ.

1. all references changed by update $\mathcal{U}$ are legal,
2. all references between objects that were already created in the pre-state of $\mathcal{U}$ remain legal and
3. all references pointing from objects, that were not yet created in the pre-state of $\mathcal{U}$ but are created in its post state, are legal (This is not covered by the first item since these references could already have existed in the pre-state without being modified by $\{\mathcal{U}\}$).
4. In addition, we have to ensure that the axioms (shown in equations (6.2) and (6.4) and used in the taclets stackInjective and maAllocOuter) constraining the attribute `stack` still hold and that
5. the implicit field <ma> is unequal **null** for all created objects as stated by equation (6.1) (rule maNonNull)

For addressing the first of the above items we have to determine which location function symbols are affected by $\{\mathcal{U}\}$ and add the corresponding axioms to Φ. As, for instance, in every reachable program state all non-static attributes different from <ma>`@Object` and `parent@MemoryArea` and unequal **null** must point to an object allocated in the same or an outer scope (as exploited by rules outerRefAttr1, outerRefAttr2 and imSub3), we have to add the axiom

$$\begin{aligned}&\{\mathcal{U}\}\forall\, T\; o;\; (o \not\doteq \mathtt{null} \wedge o.a@(T) \not\doteq \mathtt{null} \wedge o.\texttt{<c>} \doteq \mathit{TRUE}) \rightarrow\\ &o.a@(T).\texttt{<ma>}\texttt{.stack} \preceq o.\texttt{<ma>}\texttt{.stack}\end{aligned} \tag{6.5}$$

to this conjunction for each non-static attribute $a@(T)$ updated by $\mathcal{U}$. Accordingly we have to create corresponding formulas for the cases that

- array slots (we utilise the property that array slots refer only to outer scopes in reachable states in the rules outerRefArray1, outerRefArray2 and imSub4),
- static attributes (pointing to immortal memory, stated in rule staticIm),

are updated.

As item 2 states, we need to show that existing legal references have not become illegal by changes of the implicit attribute <ma> or the attribute `stack`. This is ensured by adding the following formula to Φ in case <ma> or `stack` are changed by $\{\mathcal{U}\}$:

$$\begin{aligned}&\forall\, \mathit{Object}\; o_1, o_2;\\ &\quad o_1.\texttt{<c>} \doteq \mathit{TRUE} \wedge o_2.\texttt{<c>} \doteq \mathit{TRUE} \wedge o_1.\texttt{<ma>}\texttt{.stack} \preceq o_2.\texttt{<ma>}\texttt{.stack} \rightarrow\\ &\quad \{\mathcal{U}\}(o_1.\texttt{<ma>}\texttt{.stack} \preceq o_2.\texttt{<ma>}\texttt{.stack})\end{aligned} \tag{6.6}$$

Also attributes and array slots of objects/arrays newly created by $\mathcal{U}$ need to constitute legal references (item 3). Thus we have to add formula 6.5 for each attribute $a@(T)$ of every type T for which $T.$<ntc> is changed by $\mathcal{U}$. For array slots of newly created arrays we proceed analogously.

For ensuring the constraints imposed on `stack` and <ma> (item 4 and 5) we simply add the corresponding axioms (equations (6.2), (6.4) and (6.1)) to Φ if needed (i.e., if the field `stack` or <ma> is assigned to by $\{\mathcal{U}\}$).

6.7 Rules for Symbolic Execution

For the evaluation of RTSJ programs by symbolic execution we have to take into account the runtime checks an RTSJ compliant JVM performs and the exceptions originating from this. Luckily, this applies only to `IllegalAssignmentErrors` since all other RTSJ typical exceptions we deal with are handled by the reference implementation of the RTSJ memory management API. In the following the symbolic execution rules for assignment operations necessitating such illegal-assignment runtime checks are described. These assignments can be subdivided in two classes: (i) assignments to static references and (ii) assignments to non-static references (instance attributes, array slots). In the former case the assigned object has to reside in immortal memory, in the latter case it is sufficient that it resides in the same or in a more outer scope than the object (array) whose attribute (array slot) constitutes the left-hand side of the assignment.

6.7.1 Assignments to Static References

Static reference type attributes can only reference **`null`** or objects allocated in immortal memory. This has to be checked when assigning an object to a static attribute (sa denotes a reference type static attribute and v a program variable of compatible type):

$$\textsf{sRAttrWrite}\ \frac{\begin{array}{c}\Gamma,\ \{\mathcal{U}\}(\mathsf{im}(v.\texttt{<ma>}\,\texttt{.stack}) \lor v \doteq \texttt{null}) \Rightarrow \{\mathcal{U}\}\{sa := v\}\langle \pi\, \omega\rangle\, \varphi,\ \Delta \\ \Gamma,\ \{\mathcal{U}\}(\neg\, \mathsf{im}(v.\texttt{<ma>}\,\texttt{.stack}) \land v \not\doteq \texttt{null}) \Rightarrow \{\mathcal{U}\}\langle \pi\ \texttt{IAE;}\ \omega\rangle\, \varphi,\ \Delta\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle \pi\ \texttt{sa=v;}\ \omega\rangle\, \varphi,\ \Delta}$$

In the above rule we have to distinguish two cases (indicated by the two premises):

- the assignment is legal since v is **`null`** or allocated in immortal memory. In this case the assignment is executed and the resulting state change is expressed by the update $\{sa := v\}$.

- the assignment is illegal since v is not **`null`** and not allocated in immortal memory. This leads to an `IllegalAssignmentError` to be raised. The statement

  ```
  throw new IllegalAssignmentError();
  ```

  was abbreviated with `IAE;` in the above rule.

6.7.2 Assignments to Non-Static References

For write accesses to instance attributes we have to distinguish two cases: Either the attribute assigned to is `parent@(MemoryArea)` or it is not. In the former case no check for an illegal assignment is performed since the attribute `parent` may actually refer to objects in inner scopes. As `parent` is only used for modeling purposes and does not correspond to a real attribute in `MemoryArea`, this treatment cannot lead to undetected `IllegalAssignementErrors`. In the latter case we have to take into account the possibility that the assignment raises an `IllegalAssignementError` which is modeled by the following rule (where o and v are program variables and a is a reference type attribute different from `parent@MemoryArea`):

$$\mathsf{rAttrWrite}\ \frac{\begin{array}{c}\Gamma,\ \{\mathcal{U}\}(o \not\doteq \texttt{null} \land \psi) \Rightarrow \{\mathcal{U}\}\{o.a := v\}\langle\pi\,\omega\rangle\,\varphi,\ \Delta \\ \Gamma,\ \{\mathcal{U}\}o \doteq \texttt{null} \Rightarrow \{\mathcal{U}\}\langle\pi\,\texttt{NPE;}\ \omega\rangle\,\varphi,\ \Delta \\ \Gamma,\ \{\mathcal{U}\}(o \not\doteq \texttt{null} \land \neg\psi) \Rightarrow \{\mathcal{U}\}\langle\pi\,\texttt{IAE;}\ \omega\rangle\,\varphi,\ \Delta\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle\pi\,\texttt{o.a=v;}\ \omega\rangle\,\varphi,\ \Delta}$$

where $\psi := v \doteq \texttt{null} \lor v.\text{<ma>}\texttt{.stack} \preceq o.\text{<ma>}\texttt{.stack}$ states that v is either **`null`** or allocated in the same scope as o or an outer scope of it. In the above rule we have to distinguish three cases (indicated by the three premises):

- o is not **`null`** in the state described by the update $\mathcal{U}$ and the assignment is legal with respect to the RTSJ constraints on this (i.e., when formula ψ holds),
- $o \doteq \texttt{null}$ holds and a `NullPointerException` is raised (abbreviated by `NPE;`) or
- the former two cases do not hold and an `IllegalAssignmentError` is thrown (abbreviated by `IAE;`).

Assignments to attributes without an explicit prefix (i.e., being prefixed with an implicit **`this`** reference) are handled analogously after determining the receiver object on the left-hand side from the execution context π.

When assigning to the `parent@MemoryArea` attribute, no runtime check for an illegal assignment is performed:

$$\mathsf{parentWrite}\ \frac{\begin{array}{c}\Gamma,\ \{\mathcal{U}\}o \not\doteq \texttt{null} \Rightarrow \{\mathcal{U}\}\{o.\mathit{parent} := v\}\langle\pi\,\omega\rangle\,\varphi,\ \Delta \\ \Gamma,\ \{\mathcal{U}\}o \doteq \texttt{null} \Rightarrow \{\mathcal{U}\}\langle\pi\,\texttt{NPE;}\ \omega\rangle\,\varphi,\ \Delta\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle\pi\,\texttt{o.parent@MemoryArea=v;}\ \omega\rangle\,\varphi,\ \Delta}$$

Assignments to array slots are handled analogously to assignments to instance attributes. Beside the kinds of errors we encountered in rule rAttrWrite we have to take into account two new ones now: the `ArrayIndexOutOfBoundsException` and the `ArrayStoreException`. The first one occurs (self-explanatorily) when the array is accessed with an index outside the array bounds ($[0, \mathit{length} - 1]$). The second one can be blamed on the subtyping relation of array types being covariant to the subtyping relation on its elements: $B[]$ is a strict subtype of $A[]$ if and only if B is a strict subtype of A. Thus an array arr of static type $A[]$ may have the exact type $B[]$ and must therefore not contain elements of exact type A. If an assignment operation violates this constraint by assigning an object of exact type A to an array slot of arr an `ArrayStoreException` is thrown. This is not ruled out by the type system since arr has the correct static type. Altogether we have to distinguish 5 cases:

$$\mathsf{arrayWrite}\ \frac{\begin{array}{l}\Gamma,\ \{\mathcal{U}\}(a \not\doteq \texttt{null} \land \psi_{ib} \land \psi_{av} \land \psi_{os}) \Rightarrow \{\mathcal{U}\}\{a[j] := o\}\langle\pi\,\omega\rangle\,\varphi,\ \Delta \\ \Gamma,\ \{\mathcal{U}\}a \doteq \texttt{null} \Rightarrow \{\mathcal{U}\}\langle\pi\,\texttt{NPE;}\ \omega\rangle\,\varphi,\ \Delta \\ \Gamma,\ \{\mathcal{U}\}(a \not\doteq \texttt{null} \land \neg\psi_{ib} \land \psi_{av} \land \psi_{os}) \Rightarrow \{\mathcal{U}\}\langle\pi\,\texttt{AIOOBE;}\ \omega\rangle\,\varphi,\ \Delta \\ \Gamma,\ \{\mathcal{U}\}(a \not\doteq \texttt{null} \land \psi_{ib} \land \neg\psi_{av} \land \psi_{os}) \Rightarrow \{\mathcal{U}\}\langle\pi\,\texttt{ASE;}\ \omega\rangle\,\varphi,\ \Delta \\ \Gamma,\ \{\mathcal{U}\}(a \not\doteq \texttt{null} \land \psi_{ib} \land \psi_{av} \land \neg\psi_{os}) \Rightarrow \{\mathcal{U}\}\langle\pi\,\texttt{IAE;}\ \omega\rangle\,\varphi,\ \Delta\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle\pi\,\texttt{a[j]=o;}\ \omega\rangle\,\varphi,\ \Delta}$$

where o is a reference type program variable, `AIOOBE` stands for

```
throw new ArrayIndexOutOfBoundsException()
```

and ASE stands for

```
throw new ArrayStoreException()
```

The formulas ψ_{ib}, ψ_{av} and ψ_{os} are defined as follows:

$$\begin{array}{rcl} \psi_{ib} & := & 0 \leq j \wedge j < a.length \\ \psi_{av} & := & arrayStoreValid(a, o) \\ \psi_{os} & := & o \doteq \texttt{null} \vee o.\texttt{<ma>}.\texttt{stack} \preceq a.\texttt{<ma>}.\texttt{stack} \end{array}$$

With *arrayStoreValid* being a predicate indicating that an element can legally be assigned to an array slot. There are further rules for handling *arrayStoreValid* formulas.

6.8 Related Work

This Section focused on verifying real-time JAVA programs complying (with only minor restrictions; see Sect. 5) with the existing RTSJ. Most related approaches try to improve analyzability of real-time JAVA applications by further restricting or changing the RTSJ memory model losing, in turn, some of the flexibility it provides.

Kwon and Wellings [Kwon and Wellings, 2004] describe a memory management model making use of implicitly created memory scopes associated with each method leading to something comparable to stack allocation of objects only locally used by a method. The absence of explicit scope identities, for instance, eliminates the need for enforcing the single parent rule since it is impossible to reenter a scope. `IllegalAssignemntErrors` still remain an issue to consider, but checking their absence statically is eased by the simpler memory model and can, for instance, be done by escape analysis [Choi et al., 1999].

To reduce the error-proneness of RTSJ programs several profiles for *safety critical Java* (SCJ) applications have been proposed [Kwon et al., 2005, Schoeberl et al., 2007] building upon RTSJ and imposing restrictions, for instance, on the nesting hierarchy of scopes.

Several works [Andreae et al., 2007, Boyapati et al., 2003, Zhao et al., 2004] have proposed an encoding of the nesting relation of scopes in the type system. The outlives relation between memory regions defined in [Boyapati et al., 2003] bears similarities to the relation $\succeq$ introduced in Sect. 6.3.1. One major difference is however that, unlike in [Boyapati et al., 2003], $\succeq$ represents only a snapshot of the nesting relation between scopes, thus allowing it to change during the program run.

7 JML Specifications and Proof Obligations

In this section we exemplarily consider how RTSJ programs can be specified with JML and what the resulting proof obligations are which can be discharged by the KeY prover. It is also explained what feedback can be gained from failed proofs.

7.1 JML Extensions

JML offers no possibility to talk about the nesting of scopes. In the following examples we will therefore use some newly defined JML constructs, some of which have already been stepwise introduced in the previous section. Below we give a short summary on which JML constructs were added and how they compare to their Java DL counterparts. Sections 7.1.5 and 7.1.6 show how existing and in this section newly introduced JML specification concepts can be used to define new specification constructs for conveniently specifying SCJ programs.

7.1.1 The `\currentMemoryArea` Pointer

The `\currentMemoryArea` reference points to the memory area currently active in the state the `\currentMemoryArea` expression is evaluated. When compiling a JML specification to Java DL, it is mapped to the program variable *defaultMemoryArea* introduced in Section 6.2.1:

$$T^{DL}_{JML}(\texttt{\textbackslash currentMemoryArea}) := defaultMemoryArea$$

7.1.2 The `\memoryArea` construct

The memory area an object is allocated in is determined by the implicit field `<ma>`. In JML specifications that are not aware of any implicit fields we access `<ma>` via the function `\memoryArea` with:

$$T^{DL}_{JML}(\texttt{\textbackslash memoryArea(o)}) := o.\texttt{<ma>}$$

where `o` is a reference type JML expression.

7.1.3 The `\outerScope` and `\inOuterScope` constructs

The nesting hierarchy of scopes can be described using `\outerScope`. For two memory areas `a` and `b`, `\outerScope(a, b)` evaluates to true iff `a` is an outer scope of `b` in the state the `\outerScope` expression is evaluated in. These are exactly those states s with

$$s \models a.stack \preceq b.stack$$

Therefore, `\outerScope` is translated to a Java DL formula as follows:

$$T^{DL}_{JML}(\texttt{\textbackslash outerScope(a,b)}) := a.stack \preceq b.stack$$

For convenience reasons we define the construct `\inOuterScope`, where for two reference type JML expressions `x` and `y` `\inOuterScope(x, y)` is syntactic sugar for

`\outerScope(\memoryArea(x), \memoryArea(y))`

Therefore we get:

$$T^{DL}_{JML}(\texttt{\textbackslash inOuterScope(x,y)}) := x.\texttt{<ma>}.stack \preceq y.\texttt{<ma>}.stack$$

With `\inOuterScope(x, y)` we express that the object `x` is allocated in a more outer scope (or in the same) scope compared to `y`. This is a necessary requirement for assignments of the form `y.a=x;` or `y[i]=x;` to be legal.

Example 7.1 *Setter methods cannot always be safely called in RTSJ programs:*

```
—— Java + JML ——
/*@ public normal_behavior
  @  requires \inOuterScope(a, this);
  @*/
public void setA(MyObject a){
  this.a = a;
}
—— Java + JML ——
```

If the above precondition is not met, `setA` raises an `IllegalAssignementError`.

7.1.4 The `\inImmortalMemory` Construct

The expression `\inImmortalMemory(o)` is true if and only if the object `o` is allocated in immortal memory. The translation to Java DL is again canonical:

$$T^{DL}_{JML}(\texttt{\textbackslash inImmortalMemory(o)}) := \mathsf{im}(o.\texttt{<ma>}.stack)$$

Example 7.2 *Static fields can only refer to **`null`** or objects allocated in immortal memory therefore the following specification holds:*

```
—— Java + JML ——
public class Complex{
  public static /*@nullable@*/ Complex zero;
  ...
  /*@ public normal_behavior
    @  requires zero==null ==>
    @        \currentMemoryArea==ImmortalMemory.instance();
    @  assignable zero, \object_creation(Complex);
    @  ensures \inImmortalMemory(\result);
```

```
     @*/
   public static Complex zero(){
     if(zero==null){
       zero=new Complex(0, 0);
     }
     return zero;
   }
   ...
 }
```

Java + JML

7.1.5 The arbitraryScope and arbitraryScopeThis Modifiers

By default we assume that every reference type argument of a method as well as the receiver object of a non-static method is allocated in the current or an outer scope. The effect is the same as adding for each reference type argument `a` the clause

Java + JML

```
requires \outerScope(\memoryArea(a), \currentMemoryArea);
```

Java + JML

to each specification case of the regarded method and also

Java + JML

```
requires \outerScope(\memoryArea(this), \currentMemoryArea);
```

Java + JML

in the case of an instance method.

This approach is comparable to the JML policy of treating all method arguments as non-null by default if not specified differently: like **`null`** arguments, arguments residing in inner scopes make a method error-prone since referencing them can lead to `IllegalAssignmentErrors`[1]. Thus, as for non-null arguments, the less error-prone alternative is chosen as default.

Example 7.3 *The JML method specification*

Java + JML

```
/*@ public normal_behavior
  @   ensures \result>0;
  @*/
public int m(Object a, Object b){...}
```

Java + JML

can be desugared to

[1] Such a reference does not need to be explicitly created in a program. It is sufficient for an `IllegalAssignmentError` to occur that an inner scope *final* argument is used inside a local class (see Section 7.3). The Java compiler desugars local class declarations to top-level classes containing instance fields to store the values of the final variables stemming from the enclosing block.

Java + JML

```
/*@ public normal_behavior
  @   requires \outerScope(\memoryArea(this), \currentMemoryArea);
  @   requires \outerScope(\memoryArea(a), \currentMemoryArea);
  @   requires \outerScope(\memoryArea(b), \currentMemoryArea);
  @   requires a!=null && b!=null;
  @   ensures \result>0;
  @*/
public int m(Object a, Object b){...}
```

Java + JML

For allowing arguments that reside in an arbitrary scope the modifier **arbitraryScope** can be attached to the respective argument. The modifier **arbitraryScopeThis** attached to the method declaration has the same effect for the **this** pointer.

Example 7.4 *The JML method specification*

Java + JML

```
/*@ public normal_behavior
  @   ensures \result>0;
  @*/
public /*@arbitraryScopeThis@*/ int m(
  /*@arbitraryScope@*/ Object a,
  /*@nullable@*/       Object b){...}
```

Java + JML

can be desugared to

Java + JML

```
/*@ public normal_behavior
  @   requires \outerScope(\memoryArea(b), \currentMemoryArea);
  @   requires a!=null;
  @   ensures \result>0;
  @*/
public /*@arbitraryScopeThis@*/ int m(
  /*@arbitraryScope@*/ Object a
  /*@nullable@*/       Object b){...}
```

Java + JML

7.1.6 The scopeSafe Modifier

The **scopeSafe** modifier can be attached to method or classes where the latter variant is syntactic sugar for attaching it to all methods of the considered class. The intuitive semantics of a method being declared **scopeSafe** is that is can safely be executed in scoped memory. This means that it terminates and does not raise any RTSJ memory model specific Exceptions. The **scopeSafe** modifier can be desugared to

— Java + JML —

```
/*@ public behavior
  @   signals (Throwable e)
  @     !(e instanceof javax.realtime.IllegalAssignmentError ||
  @       e instanceof javax.realtime.ScopedCycleException ||
  @       e instanceof javax.realtime.InaccessibleAreaException ||
  @       e instanceof javax.realtime.ThrowBoundaryError)
  @*/
```

— Java + JML —

Example 7.5 *The following method computing the conjugate of a complex number is, for instance, not* **`scopeSafe`***:*

— Java + JML —

```
  public Complex getConjugate(){
    if(conjugate==null){
      conjugate=new Complex(real, -im);
    }
    return conjugate;
  }
```

— Java + JML —

When executing `getConjugate()` *in a scope that differs from the scope of the receiver object and is also not an outer scope of it, the assignment*

```
 conjugate=new Complex(real, -im);
```

raises an `IllegalAssignmentError`*. In contrast, this method is* **`scopeSafe`***:*

— Java + JML —

```
  public Complex /*@scopeSafe@*/ add(Complex a){
    return new Complex(a.real+real, a.im+im);
  }
```

— Java + JML —

This does, however, not mean that it can never throw an exception (if `a==null` *holds it actually does throw a* `NullPointerException`*). The modifier* **`scopeSafe`** *only states that no exception (that is not caught internally by the method) is caused by using the RTSJ instead of the standard* Java *memory model.*

7.2 Proof Obligations for KeYSCJ Programs

EnsuresPost POs for KeYSCJ methods are created in principle analogously to the POs shown in Section 2.3.5. The *EnsuresPost* PO for a method contract $ct = (Pre, Post, Mod)$ is given by:

$$EnsuresPost(ct, Assumed) := \begin{array}{l} Conj_{Assumed} \wedge Conj_{InvRealtime} \wedge Pre \wedge ValidCall_{op}^{SCJ} \rightarrow \\ \langle PRG_{op}()\rangle Post \end{array}$$

where

- $ValidCall^{SCJ}_{op} := \begin{array}{l} ValidCall^{Java}_{op} \land \\ defaultMemoryArea \not\doteq \mathbf{null} \land \\ defaultMemoryArea.\texttt{<c>} \doteq \mathit{TRUE} \land \\ defaultMemoryArea.\mathtt{stack} \not\doteq \mathbf{null} \end{array}$

- $Conj_{InvRealtime}$ is the conjunction of the class invariants defined in `javax.realtime.*`.

- $Conj_{Assumed}$ and $ValidCall^{Java}_{op}$ are defined as in Section 2.3.5.

All other POs shown in Section 2.3.5 are adapted analogously by strengthening their precondition with $Conj_{InvRealtime}$ and $ValidCall^{Java}_{op}$.

7.3 An Extended Example

The method `foo` shown in Figure 7.1 mimics to a certain extend the behavior of a typical RT application as described in Section 4:

- During an initialisation phase the memory scopes and objects reusable in each of the mission phases are allocated. In the considered method `foo` this is the `Runnable r` and the `LTMemory m`.

- In the mission phases `r` executes in `m`. After each phase `m` is freed.

- In the mission phase itself represented by the `run` method of `r` a set of data is created (the array `a`) and some operation is performed with this data (represented by the method `bar`). The memory allocated to `a` and the instances of `A` with which it is initialised is reclaimed after each run.

The JML specification of `foo` claims that it is **`scopeSafe`** which is, however, incorrect. The reason for this incorrectness can be quickly found when trying to prove the method specification to be correct. In the prove tree one goal remains unclosable and a closer look on the open branch (see Figure 7.2) reveals that the implicit reference the `Runnable r` holds (as an instance of an inner class) to its enclosing object can cause an `IllegalAssignementError`. From looking at the open goal we can also determine that the exception is thrown in the case of

$$\neg self.\texttt{<ma>}.\mathtt{stack} \preceq defaultMemoryArea$$

holding. Thus we could either

- accept that `foo` is not scope safe and remove the **`scopeSafe`** modifier from the code. The remaining specification case

  — Java + JML —

  ```
  /*@ public normal_behavior
    @  requires x>=0 && y>=0;
    @*/
  public final void foo(){...
  ```

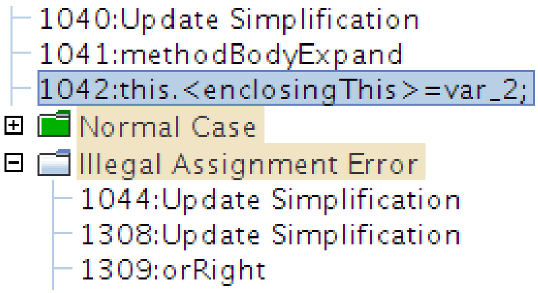
  — Java + JML —

— Java + JML —

```
import javax.realtime.*;

public abstract class B{

  /*@ public normal_behavior
    @  requires x>=0 && y>=0; @*/
  public final void /*@scopeSafe@*/ foo(final int x, final int y){
    ScopedMemory m = new LTMemory(24*x+16);
    Runnable r = new Runnable(){
      public void run(){
        int j=0;
        A[] a = new A[x];
        /*@ loop_invariant j>=0 && \object_creation(A) &&
          @   (\forall int k; k>=0 && k<j; a[k]!=null);
          @ assignable j, a[*], \object_creation(A);
          @ decreasing x-j;
          @*/
        while(j<x){
          a[j] = new A();
          j++;
        }
        bar(a);
      }
    };
    int i=0;
    /*@ loop_invariant i>=0 && \object_creation(A) &&
      @   \object_creation(A[]) && \object_creation(MemoryStack);
      @ assignable i, \object_creation(A), \object_creation(A[]),
      @   \object_creation(MemoryStack);
      @ decreasing y>0 ? y-i : 0;
      @*/
    while(i<y){
      m.enter(r);
      i++;
    }
  }

  /*@ public normal_behavior
    @  assignable \nothing;
    @*/
  public void bar(A[] a);
}
```

— Java + JML —

Figure 7.1: RTSJ code (incorrectly) specified with JML



1040:Update Simplification
1041:methodBodyExpand
1042:this.<enclosingThis>=var_2;

1044:Update Simplification
1308:Update Simplification
1309:orRight

Figure 7.2: Extract from proof tree for the **scopeSafe** PO for method `foo`

is, however, correct and does not need to be adapted since due to the absence of an **arbitraryScopeThis** modifier we implicitly require that

`\outerScope(\memoryArea(this), \currentMemoryArea)`

holds in the pre-state of `foo`.

- or define `foo` and `bar` to be static which eliminates the problem with the enclosing instance of `r` since the anonymous class is now defined within a static context and does thus not possess an enclosing instance. The **scopeSafe** modifier would then be correct.

8 Applicability to Existing SCJ Profiles and Required Adaptations

The approach described in Sections 6 and 7 is only applicable to RTSJ programs complying with the KeYSCJ profile. This can, however, be considered a minor restriction since all existing[1] SCJ profiles based on the RTSJ comply with the KeYSCJ profile.

In the Hija [HIJA, 2006, Kung et al., 2006] project a SCJ profile was proposed limiting the use of scoped memory by allowing

- only linear nesting of scopes,
- the immortal scope only to occur at the bottom of the scope stack and
- prohibiting the use of heap memory.

This use of scoped memory is obviously compatible with KeYSCJ. The Ravenscar-Java profile [Kwon et al., 2005] as well as the profile proposed in [Schoeberl et al., 2007] are even more restrictive concerning memory management allowing only one scope per thread.

Other Approaches [Kwon and Wellings, 2004, Nilsen, 2006] abstains from scopes that are explicitly created and managed by the programmer and provide scopes that are instead implicitly created when calling a method or constructor. This concept is implemented in the Java dialect supported by the commercial tool set PERC Pico [Aonix, 2008, Nilsen, 2006, Nilsen, 2008]. Since each method call is (by default[2]) associated with its own scope in which objects not outliving the method execution can be allocated, this concept bears some resemblance to stack allocation.

In [Kwon and Wellings, 2004] two possibilities to implement such a method-invocation-based (MIB) memory model were suggested:

- Based on a cross-compiler producing RTSJ code or
- based on a JVM or AOT (i.e., ahead-of-time) compiler that is aware of the memory model.

For verification of programs based on a MIB MM we have two similar choices:

- Cross-compilation to (KeYSCJ compliant) RTSJ code and verification of the generated code without modification of the approach described in the previous sections or

[1] This refers only to those SCJ profiles ([Schoeberl et al., 2007, Kung et al., 2006, Kwon et al., 2005]) which are, to the best of the author's knowledge, sufficiently well-known and considered relevant within the safety-critical Java community.

[2] The memory model described in [Kwon and Wellings, 2004] allows methods to execute in the caller's scope. Whether a method creates its own scope or depends on the caller's scope has to be specified explicitly by the programmer.

- adaptation of the calculus to directly treat the MIB memory model. Since proofs are then performed on the original source code, this alternative is preferable when it comes to interactive proving. The calculus extensions this alternative necessitates are discussed in Section 8.1.

8.1 Implicit Scopes Based on Method Invocation

Both approaches ([Kwon and Wellings, 2004] and [Nilsen, 2006]) we consider here share the following characteristics:

- Methods (can) allocate a scope (we call it *local scope* in the following) having method lifetime for the allocation of objects not outliving the method execution.

- Facilities for allocating objects in other scopes than the current local scope are provided. Both approaches realize this through Java 5 annotations indicating that certain allocations are performed in other scopes.

To handle the first item the rule methodBodyExpand for inlining method bodies is adapted in this respect that the `<cma>` pointer of the resulting local scope is set to a fresh instance of `LTMemory`.

For treating the second item we have to differentiate in which scope the new object is to be allocated. Both approaches allow allocation of objects directly in the scope of the calling method. Allocation does not necessarily need to take place in the direct caller's scope, if a returned object needs to be relayed back through a sequence of method calls this object is allocated in the original caller's scope. To mimic this behavior in the calculus we extend the *method frame* statement by another pointer pointing to the scope for allocating returned objects in.

The modifications of the calculus sketched above are discussed in more detail in Sections 8.1.1 and 8.1.2.

8.1.1 Syntactical Changes

In [Kwon and Wellings, 2004] whether a new object is allocated in the caller context is indicated by the annotation **`@ReturnedObject`** attached to the statement performing the allocation. PERC Pico [Nilsen, 2006] provides the method
`\jj{ScopeManagement.allocInCallerContext}`
for this purpose.

To keep track of caller memory areas we introduce a pointer `<oma>` (shorthand for *outer memory area*) pointing to that caller context in which returned objects are to be allocated:

Java

```
method-frame(result->retvar,
              source=T,
              this=self,
              <cma>=cm,
              <oma>=om) : {body}
```

Java

As for the `<cma>` pointer introduced in Section 6.2.1, the scope `<oma>` refers to is determined by the enclosing *method frame*. Like `<cma>`, the pointer `<oma>` can only occur inside programs (and not as a term).

Besides local scopes, PERC Pico defines two additional kinds of scopes: *constructed* and *reentrant* scopes. Both are associated with an object and have the same lifetime as this object. Since *constructed* and *reentrant* scopes have the same lifetime as the object `o`, they are associated with we abstract from the fact that they are physically different from the scope `o` is allocated in and consider the *constructed* and *reentrant* scope of `o` to be identical with `o.<ma>`. A constructed scope created within

- a method is associated with the object returned by this method.
- a constructor is associated with the object initialised by the constructor.

In both cases allocation in the constructed scope can only be performed by the method (or constructor) by which it was created. A *reentrant* scope can be seen as a special case of a *constructed* scope lacking this limitation: A *reentrant* scope can also be used for performing object allocation by instance methods of the associated object.

Whether an object needs to (for avoiding dangling references) be allocated in the *reentrant/constructed* scope of an object is determined by PERC Pico by a data flow analysis performed at compile-time. Thus the programmer is not required to explicitly specify the target scope for each object object allocation. We assume that the results of this analysis are back-annotated in the code or made available in a different way usable for program verification.

To be able to determine in which scope a method/constructor call will allocate objects outliving the method/constructor call (e.g., returned objects newly created by the called method or, in case of a constructor call, the object created by the corresponding **new** operator) we apply a program transformation adding (we refer to this transformation as T_{MIB} in the following) an annotation of the form `@scope` to each *new* operator and method call where `@scope` stands for one of the following alternatives

- the `@<oma>` pointer in case the method/constructor call will perform allocations in the caller's (of the currently executed method) scope. In the case of PERC Pico also objects allocated in the constructed scope of the returned object are considered to be allocated in the `<oma>` scope according to the abstraction we apply here for handling constructed scopes.
- the expression `@this.<ma>` in case of a method/constructor call allocating an object in the reentrant (when `@this.<ma>` occurs inside a method body) or constructed scope (when `@this.<ma>` occurs inside a constructor body)(both cases are PERC Pico specific).
- the method invocation `@ImmortalMemory.instance()` in case of a constructor invocation allocating an object in immortal memory (PERC Pico specific).
- the `@<cma>` if none of the preceding alternatives applies.

Java 5 annotations are removed by this transformation.

Example 8.1 (The Transformation $T_{\mathbf{MIB}}$) *The method* `myMethod` *is augmented with annotations introduced in [Kwon and Wellings, 2004]:*

Java

```
@ScopedMemoryMethod(...)
public MyObject myMethod(Object param){
  @ReturnedObject(type = MyObject)
  MyObject ret = new MyObject(param);
  MyObject localObj = new MyObject(ret);
  return ret;
}
```

Java

Applying the transformation T_{MIB} to the above program results in:

Java

```
public MyObject myMethod(Object param){
  MyObject ret = new@<oma> MyObject(param);
  MyObject localObj = new@<cma> MyObject(ret);
  return ret;
}
```

Java

Example 8.2 *The implicit method* `<createObject>` *(see [Beckert et al., 2007] or Section 2.3.3) is transformed as follows:*

Java

```
public static C <createObject>(){
  C newObject = C.<allocate>()@<oma>;
  newObject.<transient> = 0;
  newObject.<initialized> = false;
  newObject.<prepare>()@<oma>;
  return newObject;
}
```

Java

8.1.2 Calculus

When invoking a method possessing its own scope, a new scope has to be created and appropriately initialised. This behavior of a method invocation in a MIB MM is reflected by the rule scopedMemoryMethodBodyExpand describing the inlining of the body of a method possessing its own memory scope. This means in detail:

- The `<cma>` pointer of the newly created method frame is initialised with the repository object $get_{LTMemory}(LTMemory.\texttt{<ntc>})$.
- The scope stack of this newly created scope is initialised with a fresh scope stack $get_{MemoryStack}(j)$ for which $cma.\texttt{stack} \preceq get_{MemoryStack}(j)$ holds, where cma denotes

the memory area the `<cma>` pointer of the inner-most scope in π refers to. In other words: the local scope of the method which invokes `meth` is an outer scope of `meth`'s method local scope.

- Affected <`c`> and <`ntc`> attributes are changed adequately.

We can abstain from checking the single parent rule since scopes cannot be entered a second time due to the lack of explicit scope identities and thus this rule cannot be violated. It is also not necessary to initialise the <`ma`> attribute of the newly created scope and scope stacks since these objects are never assigned to a location different from a local variable and thus knowing their memory area is never required for evaluating a runtime check (testing for illegal assignments). Therefore, this underspecification bears no disadvantages here.

scopedMemoryMethodBodyExpand

$$\frac{\begin{array}{l} \Gamma,\ \{\mathcal{U}\}(cma.\texttt{stack} \preceq get_{MemoryStack}(j) \wedge j \geq MemoryStack.<\texttt{ntc}>) \Rightarrow \\ \quad \{\mathcal{U}\}\{get_{LTMemory}(LTMemory.<\texttt{ntc}>).\texttt{stack} := get_{LTMemory}(j) \mid\mid \\ \qquad newMem := get_{LTMemory}(LTMemory.<\texttt{ntc}>) \mid\mid \\ \qquad get_{LTMemory}(LTMemory.<\texttt{ntc}>).<\texttt{c}> := \mathit{TRUE} \mid\mid \\ \qquad LTMemory.<\texttt{ntc}> := LTMemory.<\texttt{ntc}> + 1 \mid\mid \\ \qquad MemoryStack.<\texttt{ntc}> := j + 1 \mid\mid \\ \qquad for\ i;\ if(MemoryStack.<\texttt{ntc}> \leq i \wedge i \leq j) \\ \qquad\qquad\qquad\qquad get_{MemoryStack}(i).<\texttt{c}> := \mathit{TRUE} \\ \qquad \}\langle\pi\ \texttt{method-frame(}\ \texttt{result->lhs,} \\ \qquad\qquad\qquad\qquad\quad \texttt{source=C,} \\ \qquad\qquad\qquad\qquad\quad \textbf{this}\texttt{=se,} \\ \qquad\qquad\qquad\qquad\quad \texttt{<cma>=newMem,} \\ \qquad\qquad\qquad\qquad\quad \texttt{<oma>=mem)}\ \texttt{:}\ \{\texttt{body}\}\ \omega\rangle\phi,\ \Delta \end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle\pi\ \texttt{lhs=se.meth(args)@scope@(C);}\ \omega\rangle\ \phi,\ \Delta}$$

where j is a fresh integer skolem constant (see rule push in Section 6.3.1) and mem denotes the scope the expression `scope` is resolved to. For method invocations not possessing their own local scope we just have to initialise the `<oma>` pointer with the value determined for this by the method body statement. The `<cma>` pointer is set to the value of the `<cma>` pointer of the innermost *method frame*, which we denote as *cma* here:

$$\text{methodBodyExpand}\ \frac{\begin{array}{l} \Gamma \Rightarrow \{\mathcal{U}\}\langle\pi\ \texttt{method-frame(}\ \texttt{result->lhs,} \\ \qquad\qquad\qquad\qquad\qquad \texttt{source=C,} \\ \qquad\qquad\qquad\qquad\qquad \textbf{this}\texttt{=se,} \\ \qquad\qquad\qquad\qquad\qquad \texttt{<cma>=cma,} \\ \qquad\qquad\qquad\qquad\qquad \texttt{<oma>=mem)}\ \texttt{:}\ \{\texttt{body}\}\ \omega\rangle\phi,\ \Delta \end{array}}{\begin{array}{l} \Gamma \Rightarrow \{\mathcal{U}\}\langle\ \pi\ \texttt{method-frame(...,<cma>=cma,...)}\ \texttt{:}\ \{ \\ \qquad\qquad \texttt{lhs=se.meth(args)@scope@(C);}\ p\}\ \omega\rangle\phi,\ \Delta \end{array}}$$

Due to the transformation T_{MIB} the scope an object is allocated in is determined by the annotation attached to the **`new`** operator. This is taken into account by the rule instanceCreation dispatching the constructor invocation as follows:

$$\begin{array}{c}
\textsf{instanceCreation} \\
\begin{array}{l}
\Gamma \Rightarrow \{\mathcal{U}\}\langle\pi\ \texttt{T v}_0\ =\ \texttt{T.<createObject>()@scope;} \\
\qquad\qquad \texttt{v}_0\texttt{.<init>(args)@scope;} \\
\qquad\qquad \texttt{v}_0\texttt{.<initialized>}\ =\ \textbf{\texttt{true}}\texttt{;} \\
\qquad\qquad \texttt{v}\ =\texttt{v}_0\texttt{;} \\
\qquad\quad \omega\rangle\phi
\end{array} \\
\hline
\Gamma \Rightarrow \{\mathcal{U}\}\langle\pi\,\texttt{v=new@scope T(args);}\ \omega\rangle\,\phi,\ \Delta
\end{array}$$

The scope *scope* is propagated to the called methods `<createObject>` and `<init>`. The implementation of `<createObject>` is shown in Example 8.2, the method `<init>` encapsulates the body of the called constructor.

The rule allocate representing the actual allocation of the new object also changes slightly: due to the program transformation T_{MIB} the scope the newly created object is allocated in is determined by the scope annotation attached to `<allocate>`:

$$\textsf{allocateMIB}\ \dfrac{\begin{array}{l}
\Gamma \Rightarrow \{\mathcal{U}\}\{v := get_T(T.\texttt{<ntc>})\ || \\
\qquad\qquad T.\texttt{<ntc>} := T.\texttt{<ntc>} + 1\ || \\
\qquad\qquad get_T(T.\texttt{<ntc>}).\texttt{<c>} := \mathit{TRUE}\ \ || \\
\qquad\qquad get_T(T.\texttt{<ntc>}).\,\texttt{<ma>} := mem\}\langle\pi\,\omega\rangle\,\phi,\ \Delta
\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle\pi\,\texttt{v=T.<allocate>()@scope;}\ \omega\rangle\,\phi,\ \Delta}$$

Again, *mem* denotes the scope determined by the `@scope` annotation and the execution context π.

9 A Profile for Facilitating Non-Interference Proofs in RTSJ

In a concurrent system source code verification results produced under the assumption that the code is executed in a single threaded environment can be rendered invalid. This problem is induced by concurrent (read- and write-) accesses to shared data that cause different threads to *interfere* with each other.

One means of limiting the risk of interference is to use **synchronized** blocks or methods. Synchronized code is always associated with a lockable object: If a thread tries to enter a synchronized piece of code it blocks if another thread already holds the lock for the associated object otherwise it acquires the lock and starts executing the synchronized code. Relying on synchronized code as a means for avoiding interference carries several disadvantages: (i) There is a performance overhead for locking objects and (ii) it needs to be correctly determined for which objects to acquire a lock which can be non-trivial as the following simple example shows:

— Java + JML —

```
public int computeDistance(int n){
  int dist = 0;
  for(int i=0; i<n; i++){
    dist+=read(i);
  }
  return dist;
}
```

— Java + JML —

For preventing that any thread interferes with the execution of `computeDistance` (and thus rendering any verification result for a sequential execution of this method invalid) it is not (necessarily) sufficient to declare `computeDistance` **synchronized** since this would only make the executing thread acquire the lock of the receiver object of `computeDistance`. Instead the lock for every object the result of `read` *depends on* has to be acquired to ensure that no other thread can change that data while `computeDistance` is executing.

The set of locations a method (or more general: a program) depends on is described by its *depends* clause [Müller et al., 2003]. Informally, a method `m` depending on a location `l` means that the value of `l` can have influence on the result and side effects of `m`. Complementary to the *depends* clause the *assignable* clause specifies which locations can be assigned to by a method. This two clauses can be used to determine which locks need to be acquired by the executing thread and help to identify situations in which we can do without *synchronized* code: If, for instance, n threads $t_1, \ldots, t_n$ execute n different programs $p_1, \ldots, p_n$ every program p_i with a *depends* clause d_i which is disjoint from any *assignable* clause a_j of a program p_j with $j \neq i$ we know that no other thread can interfere with the execution of p_i and thus, from the perspective of formal verification, it can be regarded as being executed sequentially.

In this context, another important specification concept facilitating data encapsulation is the *captures* clause which constrains to which objects references can be retained by executing the specified method.

Even when possessing the *depends* and *assignable* clauses of every program each thread in a system executes, one obstacle for precisely determining non-interference remains, namely aliasing: Syntactically different expressions can refer to the same location. Since deciding whether locations in, for instance, the *assignable* clause of a program `p` can be aliased by locations in the *depends* clause of a program `q` cannot be always done syntactically based only on the information provided by the respective clauses, we need additional information. Section 9.1 exemplifies how this information can be obtained by defining a profile (called KeYSCJ*) for instrumenting the existing RTSJ memory model for data encapsulation purposes to ensure non-interference.

9.1 Leveraging the RTSJ Memory Model for Data Encapsulation

The scoped memory model of RTSJ can be considered a means of data encapsulation: A Java expression `o.a@(C)` can only be an alias for `u.a@(C)` if `o` and `u` reside in the same memory area. We strengthen this already present data encapsulation mechanism of RTSJ by additionally constraining the use of scoped memory.

9.1.1 The KeYSCJ* Profile

For enhancing the data encapsulation properties of RTSJ we constrain KeYSCJ in the following way:

- The initial thread (the thread started when execution of an RTSJ program begins) possesses a dedicated scope having mission lifetime. We call this scope *mission memory* in correspondence to [HIJA, 2006]. The initial thread may only execute in immortal and mission memory.

- `Schedulable` objects may only be created and started by the initial thread and only be allocated in mission memory.

- Each `Schedulable` object (except the initial thread) must possess an initial scope[1] different from the immortal and mission memory.

- Usage of *getPortal* and *setPortal* is prohibited.

Figure 9.1 shows how a possible KeYSCJ compliant scope stack with an initial thread `T` and three spawned threads `T1`, `T2` and `T3` can look like. Note, that we do not explicitly forbid scopes to be shared, the desired data encapsulation properties are still achieved by forbidding *getPortal* and *setPortal*.

[1] Initial memory areas are provided as constructor arguments.

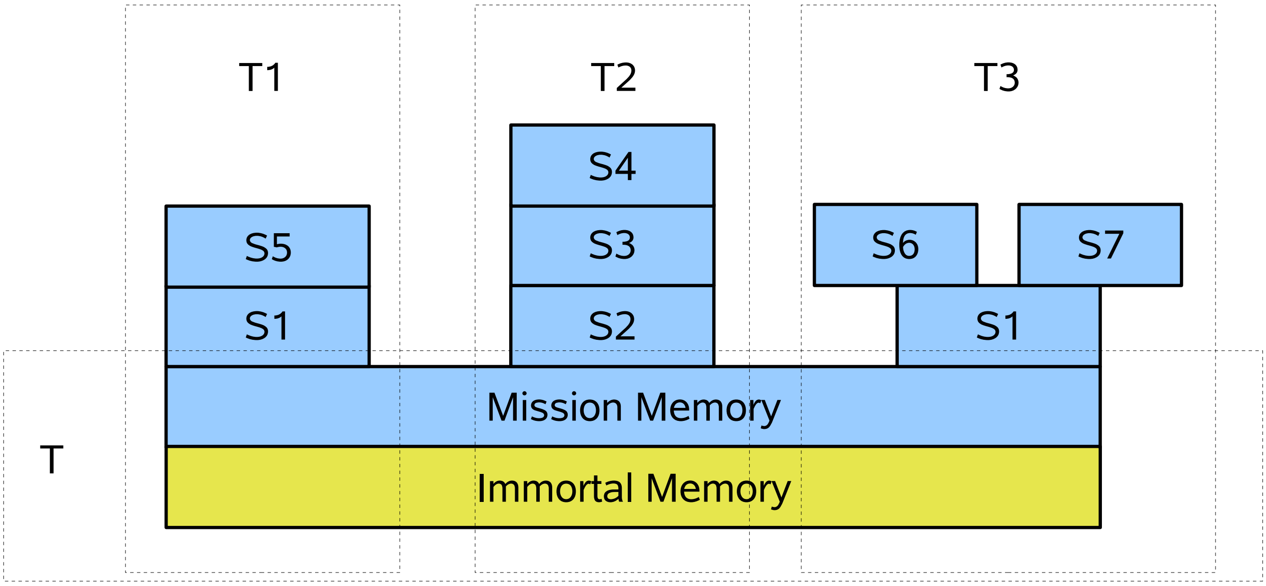


Figure 9.1: Scope stack complying with the KeYSCJ* profile. The dashed lines mark the scopes accessible by a certain thread.

9.1.2 Benefits of KeYSCJ*

Due to the restrictions made in KeYSCJ*, threads cannot share data in any scope different from the *mission* and the *immortal* scope. For determining whether a method-call can safely be performed at a certain point of the program without showing interference with other threads it is thus sufficient to check in what scopes the locations in the *assignable* and *depends* clause reside.

Lemma 9.1 *Let op be an operation, Mod its assignable clause (all locations ever assigned to by op) and Dep its depends clause (all locations ever read by op) both represented by a list of location descriptors. When executed in a KeYSCJ* compliant application by a thread T, op shows no interference with other threads if for all $l \in Mod \cup Dep$ with $l :=$ (for $\bar{x}$; if(φ) $f(t_1, \ldots, t_n)$)*

- *f is no static variable and*
- *the following holds for each reference type term t occurring in $f(t_1, \ldots, t_n)$ whose top-level function symbol is a location function symbol:*

$$\forall \bar{x}; \varphi \rightarrow (t \doteq \texttt{\textbf{null}} \vee T.getMemoryArea().\texttt{stack} \preceq t.\texttt{<ma>}.\texttt{stack})$$

where $T.getMemoryArea()$ returns the initial memory area of T.

Lemma 9.1 allows formal verification of non interference to be done (i) modularly and (ii) locally. Modularity means in this context that the implementation of called methods needs not to be taken into account, its *depends* and *assignable* clause is sufficient. Locality means here that non-interference of a program can be ensured without knowing the code executed by concurrently running threads due to the data encapsulation properties of KeYSCJ*.

9.2 Assignable, Depends and Captures Clauses

The correctness of any non-interference proof utilizing *assignable* and *depends* clauses requires, of course, correctness of the used clauses. Various works on checking the correctness of *assignable* clauses [Spoto and Poll, 2003, Sălcianu and Rinard, 2005, Cok and Kiniry, 2004] exist. A technique for proving the correctness of *assignable* clauses in KeY has been proposed in [Engel et al., 2009]. In [Bubel, 2007] a proof obligation for proving the correctness of a *depends* clause for pure methods has been presented. Checking the correctness of *captures* clauses can, for instance, be done by pointer analysis techniques [Choi et al., 1999, Whaley and Rinard, 1999, Rountev et al., 2001].

In the following a formal semantics of *assignable*, *depends* and *captures* in Java DL is presented. The proceeding Section 9.2.1 discusses how proof obligations for the correctness of these clauses can be formulated in Java DL. We only consider clauses here that contain only concrete locations and do not make use of any means of data abstraction such as *model fields* [Breunesse and Poll, 2003] and data groups [Leino, 1998]. This can be justified by the assumption that safety critical systems are naturally closed systems and thus all classes used in a Java application are known. In this setup *assignable*, *depends* or *captures* clauses containing model fields can always be desugared to equivalent clauses containing only real locations.

Definition 9.2 (Semantics of Assignable, Depends and Captures Clauses) *Let* op *be a method or constructor possessing a contract* $(Pre, Post, Mod, Dep, Cap)$*. Let further be* α *a formula containing no free variables.*

1. Mod *is a correct* assignable *clause for op under the precondition* α *if and only if for all Kripke structures* $\mathcal{K} := (\mathcal{M}, \mathcal{S}, *, \rho)$*, all states* $s, t \in \mathcal{S}$ *with* $s \models \alpha$ *and* $t := \rho(Prg_{op}())(s)$ *and all heap location function symbols* f *and all n-tuples* $a_1, \ldots, a_n$ *in the universe* $\mathbf{U}$ *the following holds*[2]*:*

$$\text{if } f^t(a_1, \ldots, a_n) \neq f^s(a_1, \ldots, a_n) \text{ then } (f, (a_1, \ldots, a_n)) \in val_s(Mod)$$

2. Dep *is a correct* depends *clause for op under the precondition* α *if and only if for all Kripke structures* $\mathcal{K} := (\mathcal{M}, \mathcal{S}, *, \rho)$*, all states* $s, t \in \mathcal{S}$ *with* $s \models \alpha$ *and* $t \models \alpha$ *and* $s \approx_{Dep} t$ *and* $s_1 := \rho(Prg_{op}())(s)$ *and* $t_1 := \rho(Prg_{op}())(t)$ *and all location function symbols* f *that are either heap location function symbols or* $f = result$ *or* $f = exc$ *(where result and exc are the program variables used in* $Prg_{op}()$ *for capturing the result of op and the exception possibly raised by it) and all n-tuples* $a_1, \ldots, a_n$ *in the universe* $\mathbf{U}$ *the following holds:*

$$\text{if } f^{t_1}(a_1, \ldots, a_n) \neq f^t(a_1, \ldots, a_n) \text{ then } f^{t_1}(a_1, \ldots, a_n) = f^{s_1}(a_1, \ldots, a_n)$$

3. Cap *is a correct* captures *clause for op under the precondition* α *if and only if for all Kripke structures* $\mathcal{K} := (\mathcal{M}, \mathcal{S}, *, \rho)$*, all states* $s, t \in \mathcal{S}$ *with* $s \models \alpha$ *and* $t := \rho(Prg_{op}())(s)$*, all reference type heap location function symbols (i.e., attributes and the array access operator)* f *and all n-tuples* $a_1, \ldots, a_n$ *in the universe* $\mathbf{U}$ *the following holds:*

$$\begin{aligned}&\text{if } f^t(a_1, \ldots, a_n) \neq f^s(a_1, \ldots, a_n) \text{ then}\\ &\quad f^t(a_1, \ldots, a_n) = val(\texttt{null}) \text{ or}\end{aligned}$$

[2] For the sake of readability, we frequently use the notation f^s in place of $s(f)$ (i.e., the interpretation of function symbol f in state s.) in this section.

$$f^t(a_1, \ldots, a_n).\texttt{<c>}^s = val(FALSE) \ \ or$$
$$there\ is\ (g, (b_1, \ldots, b_m)) \in val_s(Cap)\ such\ that\ f^t(a_1, \ldots, a_n) = g^s(b_1, \ldots, b_m)$$

Note, that the semantics of *assignable* and *depends* clauses in Java DL is more liberal than the one previously (informally) described in Section 9.1: In Java DL temporary changes of locations do not need to be reflected in the *assignable* clause, as long as the changed location is changed back to its pre-state value before the specified method terminates. In addition read access to locations needs not be reflected in the *depends* clause as long as the methods result and side effects are independent of the value of the read location. The following specification is therefore correct with respect to the Java DL semantics:

— Java + JML —

```
/*@ requires o!=null;
  @ assignable \nothing;
  @ depends \nothing;
  @*/
public void doNothing(MyObject o){
  int i = o.a++;
  o.a = i;
}
```

— Java + JML —

Remark 9.3 (*depends* Clauses in JML) *Standard JML does not know the concept of depends clauses which is a KeY specific extension. It incorporates, however, a related concept named* accessible *clause which bears slight semantic differences to the depends clause.*

Definition 9.4 (Semantics of Depends Clauses for Terms and Formulas) *Let t be a term, ϕ a formula and Dep a list of location descriptors.*

Dep is a correct depends *clause for t (and ϕ respectively) if and only if for all Kripke structures $\mathcal{K} := (\mathcal{M}, \mathcal{S}, *, \rho)$, all states $s_1, s_2 \in \mathcal{S}$ with $s_1 \approx_{Dep} s_2$ and all variable assignments β:*

$$val_{s_1,\beta}(t) = val_{s_2,\beta}(t)$$

and

$$s_1, \beta \models \phi \quad iff. \quad s_2, \beta \models \phi$$

respectively.

9.2.1 Proof Obligations for Assignable, Depends and Captures Clauses

Assignable Clauses

In [Roth, 2006] a proof obligation for *assignable* clauses was presented. A variant of this proof obligation following the same basic idea but using more recent concepts than the original one such as location dependent predicates [Bubel, 2007] (originally queries we used in a similar way) can be defined as:

$$RespectsMod(op, ct) := \{\mathcal{D}@pre(Mod)\}(\alpha \land \{*_j^{Mod@pre}\}P[*] \rightarrow [\texttt{p}]\,\{*_j^{Mod@pre}\}P[*]) \qquad (9.1)$$

where

- j is an arbitrary index such that no anonymising or anonymous update with index j occurs in α.
- $P[*]$ is an uninterpreted heap-dependent predicate.
- $\mathtt{p} := Prg_{op}()$
- $ct := (Pre, Post, Mod, Dep, Cap)$ is a contract for op.
- $\alpha := Conj_{InvRealtime} \wedge Conj_{Assumed} \wedge Pre \wedge ValidCall_{op}^{SCJ}$ with $Conj_{InvRealtime}$ and $ValidCall_{op}^{SCJ}$ as defined in Section 7.2.

Theorem 3 (Soundness and Completeness of $RespectsMod$**)** *Let op be an operation and* $ct := (Pre, Post, Mod, Dep, Cap)$ *a contract for op. The formula* $RespectsMod(op, ct)$ *is valid if and only if* Mod *is a correct* assignable *clause for op under the precondition* α*.*

A correctness and completeness proof for a slightly different variant of this PO (using formulas and rigid constants for memorizing the pre-state instead of updates) can be found in [Engel et al., 2009]. This proof can be easily adapted to the PO $RespectsMod$.

Remark 9.5 *Even though we consider proof obligations for SCJ programs in this section, the corresponding POs for standard* Java *programs can be easily obtained by setting the assumptions* α *used in the above as well as the following POs to*

$$\alpha := Conj_{Assumed} \wedge Pre \wedge ValidCall_{op}^{Java}$$

Depends Clauses

A sound PO for verifying the *depends* clause of a query (or a *pure* method in the JML jargon) is defined in [Bubel, 2007]:

$$\begin{aligned} RespectsDep_q(q, ct) \quad := \quad & \forall x, y; (\{*_j^{Dep}\}\alpha \wedge \{*_i\}\{*_j^{Dep}\}\alpha \wedge \\ & \{*_j^{Dep}\}[\mathtt{r=q;}]\, r \doteq x \wedge \{*_i\}\{*_j^{Dep}\}[\mathtt{r=q;}]\, r \doteq y \rightarrow x \doteq y) \end{aligned} \quad (9.2)$$

Where α is defined as above. The intuitive meaning of $RespectsDep_q(q, ct)$ is that if the query is called in two arbitrary states s and t with $s \approx_{Dep} t$ (s and t are "constructed" by the updates $\{*_j^{Dep}\}$ and $\{*_i\}\{*_j^{Dep}\}$) its result is the same in both states. Thus if $RespectsDep_q$ is valid, Dep is a correct dependency clause for q. Even though this PO is sound, it is utterly incomplete as soon as Dep consists of more than one location descriptor as the following simple example shows.

Example 9.6 *Let us consider the following method*

— *Java + JML* —

```
/*@ public normal_behavior
  @  depends o, o.a;
  @*/
public /*@pure@*/ int m(MyObject o){
  if(o==null) return 0;
  return o.a;
}
```

— *Java + JML* —

The depends clause of this query is obviously correct, though $\mathit{RespectsDep}_q(m(o), ct)$ *(with* ct *being the contract as shown above) is invalid.*

The reason for this invalidity becomes clear when we compare the evaluation of the depends *clause in the states reached by the updates* $\{*_i\}\{*_j^{Dep}\}$ *and* $\{*_j^{Dep}\}$*. Let us assume* $\mathit{RespectsDep}_q(m(o), ct)$ *is evaluated in a state* s_1 *let further be*

$$\begin{array}{rcl} pre_1 & := & val_{s_1}(*_j^{Dep})(s_1) \\ s_2 & := & val_{s_1}(*_i)(s_1) \\ pre_2 & := & val_{s_2}(*_j^{Dep})(s_2) \end{array}$$

Thus pre_1 *and* pre_2 *denote the states in which the query is evaluated. Then*

$$\begin{array}{rl} & val_{pre_1}(o.a) \\ = & val_{s_1}(\{*_j^{Dep}\}o.a) \\ = & \begin{cases} val_{*(j)}(o.a) & \text{if } val_{*(j)}(o) = val_{s_1}(o) \\ val_{s_1}((\{*_i\}o).a) & \text{if } val_{*(j)}(o) \neq val_{s_1}(o) \end{cases} \end{array}$$

and

$$\begin{array}{rl} & val_{pre_2}(o.a) \\ = & val_{s_2}(\{*_j^{Dep}\}o.a) \\ = & \begin{cases} val_{*(j)}(o.a) & \text{if } val_{*(j)}(o) = val_{s_2}(o) \\ val_{s_2}((\{*_i\}o).a) & \text{if } val_{*(j)}(o) \neq val_{s_2}(o) \end{cases} \end{array}$$

This shows that the location o.a can evaluate to different values in the states pre_1 *and* pre_2 *in which the query is evaluated. Therefore, we can find a witness Kripke structure in which we choose the states* s_1*,* $*(j)$ *and* $*(i)$ *in such a way that* $val_{pre_1}(o.a) \neq val_{pre_2}(o.a)$ *(and thus* $pre_1 \not\approx_{Dep} pre_2$*) which entails*

$$val_{\rho(r=m(o);)(pre_1)}(r) \neq val_{\rho(r=m(o);)(pre_2)}(r)$$

and consequently $s_1 \not\models \mathit{RespectsDep}_q(m(o), ct)$*.*

The problem arises since we only ensure that the locations to which o.a evaluates in the states s_1 *and* s_2 *(which are* $(a, (val_{s_1}(o)))$ *and* $(a, (val_{s_2}(o)))$*) are set to the same value (by update* $\{*_j^{Dep}\}$*). Since, however, the value of location o is changed as well by* $\{*_j^{Dep}\}$*, o.a can evaluate to a different location in state* pre_1 *(and* pre_2*) compared to* s_1 *(and* s_2*). In short, we set the locations to which the depends clause evaluates in the state* s_1 *and* s_2 *to identical values instead we should ensure that the locations to which it evaluates in* pre_1 *and* pre_2 *(since these are the states in which we evaluate the query) have identical values.*

Before we try to define a sound *and* complete PO for the *depends* clause we briefly recall what has to be proven: Whenever we execute a query q in two states s and t with $s \approx_{Dep} t$ (where Dep is q's *depends* clause) then the result yielded from q's execution is the same both times. The question is now how to obtain such two states if constructing them by means of updates bears such subtle pitfalls as example 9.6 illustrates. An obvious possibility is, to choose a declarative instead of constructive (as done in $\mathit{RespectsDep}_q$) approach to define these two states:

$$\begin{array}{rcl} \mathit{RespectsDep}^*_q(q, ct) & := & \forall x, y; (\alpha \wedge \{*_i\}\alpha \wedge \bigwedge_{ld:ld\in Dep} \xi_i(ld) \wedge \\ & & [\texttt{r=q;}]\, r \doteq x \wedge \{*_i\}[\texttt{r=q;}]\, r \doteq y \rightarrow x \doteq y) \end{array} \quad (9.3)$$

where

$$\xi_i(ld) := \forall \bar{x}; ((\{*_i\}\varphi \leftrightarrow \varphi) \land (\varphi \rightarrow \{*_i\}f(\bar{t}) \doteq f(\bar{t})))$$

with $ld :=$ (for $\bar{x}$; if(φ) $f(\bar{t})$). Obviously, for every state s the property $s \approx_{Dep} val_s(*_i)(s)$ holds if and only if

$$s \models \bigwedge_{ld:ld \in Dep} \xi_i(ld) \tag{9.4}$$

holds.

Theorem 4 (Soundness and Completeness of $RespectsDep_q^*$**)** *Let q be a query possessing a contract $ct = (Pre, Post, Mod, Dep, Cap)$. The formula $RespectsDep_q^*(q, ct)$ is valid if and only if Dep is a correct depends clause under the precondition α.*

Theorem 4 follows directly from equation (9.4).

Analogously we can define proof obligations for *depends* clauses of terms and formulas which do not contain virtual location function symbols:

$$DependsOn(t, Dep) \;:=\; \bigwedge_{ld:ld \in Dep} \xi_i(ld) \rightarrow \{*_i\}t \doteq t \tag{9.5}$$

$$DependsOn(\phi, Dep) \;:=\; \bigwedge_{ld:ld \in Dep} \xi_i(ld) \rightarrow \{*_i\}\phi \leftrightarrow \phi \tag{9.6}$$

Theorem 5 (Soundness and Completeness of $DependsOn$**)** *Let t be a term, ϕ a formula and Dep a list of location descriptors. Let further t and ϕ be free of virtual location function symbols. Then $DependsOn(t, Dep)$ is valid if and only if Dep is a correct depends clause for t and $DependsOn(\phi, Dep)$ is valid if and only if Dep is a correct depends clause for ϕ.*

Theorem 5 is also trivially entailed by equation (9.4).

Remark 9.7 (Comparison to other POs for depends clauses of formulas) *In previous works ([Bubel, 2007, Roth, 2006]) POs for depends clauses of formulas differing from DependsOn have already been suggested. This raises the question about the benefit of yet another PO for the correctness of such a depends clause. In short, this benefit is that DependsOn is complete with respect to our definition of depends clauses (see Definition 9.2), which is not the case for the POs introduced in the cited works as we elaborate in the following:*

The PO $po_{p[ld]}$ presented in [Bubel, 2007] is afflicted with the same completeness issue as $RespectsDep_q$: it cannot canonically be raised to a PO for sets of location descriptors without becoming incomplete.

In [Roth, 2006] the following PO[3] for proving the correctness of a depends clause Dep for a formula ϕ was introduced:

$$CorrectDepends(Dep, \phi) := \{\mathcal{D}@pre'(Dep)\}(\phi \rightarrow \{*_i\}\{\mathcal{U}\}\phi)$$

[3] This is not the actual PO introduced in [Roth, 2006] but a logically equivalent one using an update instead of a set of axioms for defining the values of the required @*pre*-functions.

with

$$\begin{array}{rcl} \mathit{Dep} & := & \{ld_1, \ldots, ld_n\} \\ \mathcal{U} & := & \{u_{ld_1} || \ldots || u_{ld_n}\} \\ u_{(for\ \bar{x};\ if(\varphi)\ f(\bar{t}))} & := & for\ \bar{x};\ if(b_{\varphi}^{@pre}(\bar{x}) \doteq \mathit{TRUE})\ f(\bar{t}^{@pre}) := f^{@pre}(\bar{t}^{@pre}) \end{array}$$

Basically the update $\mathcal{U}$ sets the locations contained in Dep in the pre-state (defined by update $\mathcal{D}@pre'(Dep)$) to their pre-state values. CorrectDepends is only complete (with respect to the semantics of depends clauses chosen in this work) for sets of location descriptors Dep and formulas ϕ complying with the following property[4]*:*

> *for all $ld := (for\ \bar{x};\ if(\varphi)\ f(t_1, \ldots, t_n))$ and all states s:*
>
> *if $ld \in Dep$ and $Dep \setminus \{ld\}$ is not a correct depends clause for ϕ then*
>
> $$val_{s,\beta}((for\ \bar{x};\ if(\varphi)\ t)) \subseteq val_{s,\beta}(Dep)$$
>
> *for each term t occurring in ld whose top-level function symbol is a location function symbol.*

Illustratively, this means, for instance, that

$$\begin{array}{l} CorrectDepends(\{o.a\}, o.a \doteq 0) = \\ \{\mathcal{D}@pre'(\{o.a\})\}(o.a \doteq 0 \rightarrow \{*_i\}\{o^{@pre}.a := a^{@pre}(o^{@pre})\}o.a \doteq 0) \end{array}$$

is (obviously) not valid even though $\{o.a\}$ is a correct depends clause for $o.a \doteq 0$ according to Definition 9.2. Adding o (which can be done since underspecifying a depends clause is viable) to the depends clause fixes this problem since

$$\begin{array}{l} CorrectDepends(\{o.a, o\}, o.a \doteq 0) = \\ \{\mathcal{D}@pre'(\{o.a, o\})\}(o.a \doteq 0 \rightarrow \{*_i\}\{o^{@pre}.a := a^{@pre}(o^{@pre}) || o := o^{@pre}\}o.a \doteq 0) \end{array}$$

is valid. This fix carries, however, the disadvantage of an overapproximated (and thus less precise) depends clause.

Generalizing this approach in a canonical way for applying it to operations with potential side effects leaves us to the following PO, in which we make use of the location dependent predicates for capturing the side effects of the regarded program:

$$\begin{array}{rcl} RespectsDep(op, ct) & := & \alpha \wedge \{*_i\}\alpha \wedge \{\mathcal{D}@pre(Mod)\}[\mathtt{p}]\,P[M] \wedge \bigwedge_{ld:ld \in Dep} \xi_i(ld) \rightarrow \\ & & \{*_i\}\{\mathcal{D}@pre(Mod)\}\langle \mathtt{p} \rangle\,P[M] \end{array} \qquad (9.7)$$

where $\{*_i\}$ is a new anonymous update, $M := (Mod_{ext}^{@pre}, result, exc)$ ($result$ is optional in case of *op* being non-void) and *ct*, $\xi_i(ld)$, $result$, exc, $\mathtt{p}$ and α are defined as before.

[4] The semantics for depends clauses used in [Roth, 2006] entails that this property is met by a correct depends clause. Thus *CorrectDepends* is complete with respect to the semantics of depends clauses defined in [Roth, 2006].

Theorem 6 (Soundness of *RespectsDep***)** *Let $ct := (Pre, Post, Mod, Dep, Cap)$ be a contract for an operation op. Then the following implication holds: If $RespectsDep(op, ct)$ is valid and Mod is a correct assignable clause under the precondition*

$$\alpha := Conj_{InvRealtime} \wedge Conj_{Assumed} \wedge Pre \wedge ValidCall_{op}^{SCJ}$$

then Dep is a correct depends clause for op under the precondition α and op terminates (when invoked in a state satisfying α).

Before we start to sketch a proof for Theorem 6 we need a helper Lemma.

Lemma 9.8 *Let $\mathcal{K} := (\mathcal{M}, \mathcal{S}, *, \rho)$ be a Kripke structure and $s, t \in \mathcal{S}$. Then for every $i \in \mathbb{N}$ there is a Kripke structure $\mathcal{K}' := (\mathcal{M}, \mathcal{S}, *', \rho)$ (coinciding with $\mathcal{K}$ except for the functions $*$ and $*'$) such that*

$$val_{\mathcal{K}',s}(*'_i)(s) = t'$$

with $t' \approx_* t$.

This Lemma follows directly from the semantics of anonymous update (see Section 2.2.1): There is a Kripke structure $\mathcal{K}' := (\mathcal{M}, \mathcal{S}, *', \rho)$ such that

$$*'(i) = t$$

Proof 2 (of Theorem 6) *The termination argument is obvious.*

For proving that the validity of $RespectsDep(op, ct)$ also implies the correctness of Dep we assume that Dep is an incorrect depends clause and then show that this entails invalidity of $RespectsDep(op, ct)$.

Let Dep be an incorrect depends clause for op. Then (according to definition 9.2) there are states s and t with

$$\begin{array}{rcl} s & \models & \alpha \quad (9.8) \\ t & \models & \alpha \quad (9.9) \\ s & \approx_{Dep} & t \quad (9.10) \end{array}$$

and $s_1 := \rho(p)(s)$ and $t_1 := \rho(p)(t)$ and either a heap location $(f, (a_1, \ldots, a_n))$ or $f \in \{result, exc\}$ (and $n = 0$) such that

$$\begin{array}{rcl} f^{t_1}(a_1, \ldots, a_n) & \neq & f^{t}(a_1, \ldots, a_n) \quad (9.11) \\ f^{t_1}(a_1, \ldots, a_n) & \neq & f^{s_1}(a_1, \ldots, a_n) \quad (9.12) \end{array}$$

As a consequence of Lemma 9.8 we assume wlog.[5] *that*

$$val_s(*_i)(s) = t \quad (9.13)$$

This implies

$$s \models \bigwedge_{ld:ld \in Dep} \xi_i(ld) \quad (9.14)$$

[5] This can be done since due to Lemma 9.8 there is a Kripke structure $\mathcal{K} = (\mathcal{M}, \mathcal{S}, *, \rho)$ such that $*(i) = t$ and thus $val_s(*_i)(s) = t'$ with $t' \approx_* t$ and thus $t' \approx_{Dep} t$ and (since $\approx_{Dep}$ is an equivalence relation) $s \approx_{Dep} t'$. This entails that $t \models \alpha$ and for $s_1 := \rho(p)(s)$ and $t_1 := \rho(p)(t')$ also equations (9.11) and (9.12) hold.

In addition we define

$$s_1 \models P[M] \tag{9.15}$$

$$t_1 \not\models P[M] \tag{9.16}$$

which is admissible according to the semantics of location dependent symbols (see Definition 2.23) since either

1. *$val_s(Mod) \neq val_t(Mod)$: Thus $val_{\rho(p)(s)}(Mod^{@pre}) \neq val_{\rho(p)(t)}(Mod^{@pre})$ which entails $s_1 \not\approx'_{Mod^{@pre}} t_1$ and due to Lemma 2.21 $s_1 \not\approx_{Mod^{@pre}_{ext}} t_1$ and therefore also $s_1 \not\approx_M t_1$. Therefore, we can legally pick the interpretations s_1 and t_1 as done in equations (9.15) and (9.16).*

2. *or $val_s(Mod) = val_t(Mod)$: Therefore, $val_{s_1}(Mod^{@pre}) = val_{t_1}(Mod^{@pre})$ and consequently $val_{s_1}(M) = val_{t_1}(M)$. From equation (9.11) and Mod being a correct assignable clause we get*

   $$(f, (a_1, \ldots, a_n)) \in \{val_t(Mod), result, exc\}$$

   and also $(f, (a_1, \ldots, a_n)) \in \{val_t(Mod^{@pre}), result, exc\}$. Together with equation (9.12) and Lemma 2.21 this entails

   $$s_1 \not\approx_M t_1$$

   Since there are no restrictions on the interpretation of $P[M]$ in states s_1 and t_1 with $s_1 \not\approx_M t_1$, the interpretations s_1, t_1 as constrained in equations (9.15) and (9.16) are again admissible.

This entails that for s the following holds:

$$s \models \alpha \wedge \{*_i\}\alpha \wedge \bigwedge_{ld:ld \in Dep} \xi_i(ld) \wedge \{\mathcal{D}@pre(Mod)\}[p]\, P[M] \tag{9.17}$$

$$s \not\models \{*_i\}\{\mathcal{D}@pre(Mod)\}\langle p \rangle\, P[M] \tag{9.18}$$

and thus

$$s \not\models RespectsDep(op, ct)$$

□

Remark 9.9 (Incompleteness of *RespectsDep***)** *The presented proof obligation for the depends clause is again not complete. This means that correctness of Dep (and Mod) does not entail validity of $RespectsDep(op, ct)$.*

The problem arises when Mod is underspecified (i.e., overapproximated). This is illustrated by the following example: Let op be a void method with an empty body, $Dep := \emptyset$, $Mod := *$ and $\alpha := true$. Then $RespectsDep(op, ct)$ is logically equivalent to:

$$P[*, exc] \rightarrow \{*_i\}P[*, exc]$$

which is obviously not valid.

For obtaining a sound *and* complete proof obligation we need to cope with underspecified *assignable* clauses. One solution could be to check for every location in the *assignable* clause if it was actually changed compared to its pre-state s and then to require only for those locations that their interpretation is the same in the post-states s' and t' (with s,s' and t' as defined above). A PO reflecting this considerations could be:

$$\mathit{RespectsDep}^*(op, ct) := \alpha' \rightarrow \{\mathcal{U}_1\}\langle \mathtt{p}\rangle\, \{\mathcal{U}_2\}\langle \mathtt{p}\rangle\, \phi(Mod^{@pre'_2}, Mod^{@pre_3}, Mod^{@pre'_3}) \tag{9.19}$$

with

- $\alpha' := \alpha \wedge \{*_i\}\alpha \wedge \bigwedge\limits_{ld:ld\in Dep} \xi_i(ld)$ and α and $\xi_i(ld)$ as defined above.
- $ct := (Pre, Post, Mod, Dep, Cap)$
- $\{\mathcal{U}_1\} := \{\mathcal{D}@pre_1(Mod)\}$
- $\{\mathcal{U}_2\} := \left\{\begin{array}{l} \mathcal{D}@pre'_2(Mod^{@pre_1})|| \\ exc' := exc|| \\ result' := result \end{array}\right\}\{*_i\}\left\{\begin{array}{l} \mathcal{D}@pre_3(Mod)|| \\ \mathcal{D}@pre'_3(Mod) \end{array}\right\}$
- exc' and $result'$ (in case op is not void) are fresh virtual location function symbols

and

$$\begin{array}{l} \phi(Mod^{@pre'_2}, Mod^{@pre_3}, Mod^{@pre'_3}) := result' \doteq result \wedge exc' \doteq exc \wedge \\ \bigwedge\limits_{m:m\in Mod^{@pre_3}} (\forall \bar{x}_m; (\varphi_m^{@pre_3} \rightarrow \left(\begin{array}{l} f_m(\bar{t}_m^{@pre_3}) \doteq f_m^{@pre_3}(\bar{t}_m^{@pre_3}) \vee \\ f_m(\bar{t}_m^{@pre_3}) \doteq f_m^{@pre_2}(\bar{t}_m^{@pre_3}) \end{array}\right))) \end{array} \tag{9.20}$$

where $m :=$ (for $\bar{x}_m$; if$(\varphi_m^{@pre_3})$ $f_m(\bar{t}_m^{@pre_3})$). To understand what $\mathit{RespectsDep}^*(p, ct)$ states we walk through it step by step (and from left to right):

- We only consider initial states that satisfy the assumptions α' (indicated by the implication $\alpha' \rightarrow \dots$), for all other states $\mathit{RespectsDep}^*(p, ct)$ is trivially true. This means in particular that $s \models \mathit{RespectsDep}^*(p, ct)$ holds trivially for all states s with $s \not\approx_{Dep} t$ for all t with $t \approx_* *(i)$ (due to the subformula $\bigwedge\limits_{ld:ld\in Dep} \xi_i(ld)$ in conjunction α').
- The *locations* Mod denotes in this state are memorized by $\{\mathcal{D}@pre_1(Mod)\}$.
- In the state reached after the execution of p the *values* of the locations in $Mod^{@pre_1}$ are memorized by $\{\mathcal{D}@pre'_2(Mod^{@pre_1})\}$. We need this information later to check whether the locations changed by p are always changed in the same way. Also the values of the program variables $result$ and exc are memorized.
- Afterwards the entire heap is anonymised by update $\{*_i\}$.
- Again we memorize the locations in Mod as well as the values of these locations in the current state by the update $\{\mathcal{D}@pre_3(Mod)||\mathcal{D}@pre'_3(Mod)\}$.

- In the state reached after the (second) execution of p we compare each location contained in $Mod^{@pre_3}$ with its pre-state (the state before the second execution of p) value. If its value changed it must be the same as in the state after the first execution of p (which was memorized by $\{\mathcal{D}@pre'_2(Mod^{@pre_1})\}$). It is also determined whether the returned result (in case of a non-void method) and the possibly thrown exception are identical.

  This is expressed by the formula $\phi(Mod^{@pre'_2}, Mod^{@pre_3}, Mod^{@pre'_3})$.

Theorem 7 (Soundness and Completeness of $\mathit{RespectsDep}^*$**)** *Let*

$$ct = (Pre, Post, Mod, Dep, Cap)$$

be a contract for op and Mod a correct assignable clause under the precondition

$$\alpha := Conj_{InvRealtime} \wedge Pre \wedge ValidCall_{op}^{SCJ}$$

The formula $\mathit{RespectsDep}^*(op, ct)$ *is valid if and only if Dep is a correct depends clause under the precondition* α *.*

Proof 3 (Soundness and Completeness of $\mathit{RespectsDep}^*(op, ct)$**)** *The soundness proof is performed in parts analogously to the proof of Theorem 6. For proving completeness we show that invalidity of* $\mathit{RespectsDep}^*(op, ct)$ *implies incorrectness of Dep with respect to op.*

As in Proof 2, we again assume incorrectness of Dep and show that this entails invalidity of formula $\mathit{RespectsDep}^*(op, ct)$*.*

Let Dep be an incorrect depends clause for op. Then (according to definition 9.2) there are states s and r with

$$s \models \alpha \tag{9.21}$$

$$r \models \alpha \tag{9.22}$$

$$s \approx_{Dep} r \tag{9.23}$$

and $s_1 := \rho(p)(s)$ *and* $r_1 := \rho(p)(r)$ *and a heap location* $(f, (a_1, \ldots, a_n))$ *or* $f \in \{result, exc\}$ *(and* $n = 0$*) such that*

$$f^{r_1}(a_1, \ldots, a_n) \neq f^{r}(a_1, \ldots, a_n) \tag{9.24}$$

$$f^{r_1}(a_1, \ldots, a_n) \neq f^{s_1}(a_1, \ldots, a_n) \tag{9.25}$$

Again we assume wlog. (due to Lemma 9.8) that

$$val_s(*_i)(s) = r \tag{9.26}$$

This implies

$$s \models \bigwedge_{ld:ld \in Dep} \xi_i(ld) \tag{9.27}$$

and together with equations (9.21) and (9.22)

$$s \models \alpha' \tag{9.28}$$

As the modifier set Mod is correct,

$$(f, (a_1, \ldots, a_n)) \in \{val_r(Mod), result, exc\} \tag{9.29}$$

and thus

$$(f,(a_1,\ldots,a_n)) \in \{val_{r_1}(Mod^{@pre_3}), result, exc\} \tag{9.30}$$

Due to equation (9.30), there is a location descriptor $ld := (\textit{for } \bar{x};\ \textit{if}(\varphi)\ f(\bar{t}))$ *and a variable assignment* β *such that*

$$\begin{aligned} r_1,\beta &\models \varphi^{@pre_3} && (9.31)\\ (f,(a_1,\ldots,a_n)) &= (f,(val_{r_1,\beta}(\bar{t}^{@pre_3}))) && (9.32)\end{aligned}$$

then due to equation (9.24)

$$r_1,\beta \not\models f(\bar{t}^{@pre_3}) \doteq f^{@pre_3}(\bar{t}^{@pre_3}) \tag{9.33}$$

and due to equation (9.25) (since $r_1(f^{@pre_2}) = s_1(f)$*)*

$$r_1,\beta \not\models f(\bar{t}^{@pre_3}) \doteq f^{@pre_2}(\bar{t}^{@pre_3}) \tag{9.34}$$

Equations (9.30) through (9.34) imply now

$$r_1 \not\models \phi(Mod^{@pre'_2}, Mod^{@pre_3}, Mod^{@pre'_3}) \tag{9.35}$$

and thus (together with (9.28))

$$s \not\models RespectsDep^*(op,ct) \tag{9.36}$$

Let us now assume $RespectsDep^*(p,ct)$ *is invalid. Then there is a Kripke structure* $\mathcal{K} := (\mathcal{M},\mathcal{S},*,\rho)$ *and states* s,s',s_1,r,r_1 *with* $s := val_{s'}(\mathcal{U}_1)(s')$, $s_1 := \rho(p)(s)$, $r := val_{s_1}(\mathcal{U}_2)(s_1)$ *and* $r_1 := \rho(p)(r)$ *such that*

$$\begin{aligned} s' &\models \alpha' && (9.37)\\ r_1 &\not\models \phi(Mod^{@pre'_2}, Mod^{@pre_3}, Mod^{@pre'_3}) && (9.38)\end{aligned}$$

Equation (9.37) implies

$$s \models \alpha' \tag{9.39}$$

since $\mathcal{U}_1$ *only assigns to virtual locations on which* α' *does not depend. Since* $s \models \bigwedge_{ld:ld\in Dep} \xi_i(ld)$ *holds (due to equation (9.39)), we get*

$$s \approx_{Dep} val_s(*_i)(s) \tag{9.40}$$

and consequently (since $r \approx_* val_s(*_i)(s)$*)*

$$s \approx_{Dep} r \tag{9.41}$$

Together with equation (9.38) we conclude from this that there is a

$$(f,(a_1,\ldots,a_n)) \in \{val_r(Mod), result, exc\}$$

such that

$$\begin{aligned} f^{r_1}(a_1,\ldots,a_n) &\neq f^{r}(a_1,\ldots,a_n) && (9.42)\\ f^{r_1}(a_1,\ldots,a_n) &\neq f^{s_1}(a_1,\ldots,a_n) && (9.43)\end{aligned}$$

Equations (9.42) and (9.43) together indicate that Dep is an incorrect depends clause with respect to op. □

Example 9.10 (*RespectsDep* and *RespectsDep∗*) *Let us revisit the method m and its specification shown in Example 9.6. Its JML specification corresponds to the contract*

$$ct := (true, true, \emptyset, \{o, o.a\}, *)$$

Thus we get the proof obligations

$$RespectsDep(m, ct) = \begin{array}{l} \alpha \wedge \{*_i\}\alpha \wedge [Prg_m()]P[result, exc] \wedge \{*_i\}o \doteq o \wedge \{*_i\}o.a \doteq o.a \rightarrow \\ \{*_i\}[Prg_m()]P[result, exc] \end{array}$$

and

$$\begin{array}{l} RespectsDep^*(m, ct) = \\ \quad \alpha \wedge \{*_i\}\alpha \wedge \{*_i\}o \doteq o \wedge \{*_i\}o.a \doteq o.a \rightarrow \\ \quad \langle Prg_m()\rangle\{exc' := exc || result' := result\}\{*_i\}\langle Prg_m()\rangle result \doteq result' \wedge exc \doteq exc' \end{array}$$

Since m is a query, we can also apply PO RespectsDep$_q^*$:

$$RespectsDep_q^*(m, ct) = \begin{array}{l} \forall x, y; (\alpha \wedge \{*_i\}\alpha \wedge \{*_i\}o \doteq o \wedge \{*_i\}o.a \doteq o.a \wedge \\ [r = m(o);]\, r \doteq x \wedge \{*_i\}[r = m(o);]r \doteq y \rightarrow x \doteq y) \end{array}$$

Captures Clauses

A proof obligation for the *captures* clause can be developed in a similar manner as for the *depends* clause: A *captures* clause for a method or constructor *op* is correct if after *op*'s execution each reference type location in the *assignable* clause (assuming the correctness of the *assignable* clause) that changed either references a value contained in the *captures* clause, is **`null`** or newly created. This is formalized by the proof obligation:

$$RespectsCap(op, ct) := \alpha \rightarrow \{\begin{array}{c} \mathcal{D}@pre'(Mod)|| \\ \mathcal{D}@pre'(Cap) \end{array}\}[\mathrm{p}]\, \psi(Mod^{@pre}, Mod^{@pre'}, Cap^{@pre'}) \quad (9.44)$$

where α is defined as above and

$$\begin{array}{l} \psi(Mod^{@pre}, Mod^{@pre'}, Cap^{@pre'}) := \\ \quad \bigwedge_{\substack{m: m \in Mod^{@pre} \\ ET(m)\ is\ ref.\ type}} (\forall \bar{x}_m; (\ \varphi_m^{@pre} \rightarrow \left(\begin{array}{l} f_m(\bar{t}_m^{@pre}) \doteq f_m^{@pre}(\bar{t}_m^{@pre}) \vee \\ f_m(\bar{t}_m^{@pre}) \doteq \mathtt{\mathbf{null}} \vee \\ f_m(\bar{t}_m^{@pre}).\mathtt{<c>}^{@pre} \doteq FALSE \vee \\ \bigvee_{\substack{c: c \in Cap^{@pre'} \wedge \\ ET(c), ET(m) \\ \text{are compatible}}} \exists \bar{x}_c; (\varphi_c \wedge f_m(\bar{t}_m^{@pre}) \doteq f_c^{@pre}(\bar{t}_c^{@pre})) \end{array} \right))) \end{array} \quad (9.45)$$

where

- $m :=$ (for $\bar{x}_m$; if$(\varphi_m^{@pre})$ $f_m(\bar{t}_m^{@pre})$) and $c :=$ (for $\bar{x}_c$; if$(\varphi_c^{@pre})$ $f_c^{@pre}(\bar{t}_c^{@pre})$)
- $ET(m)$ denotes the static type of elements of m, i.e., the static type of f_m
- we call two types T_1, T_2 compatible here if there is a Java type $T \in \mathcal{T}_d$ with $T \sqsubseteq T_1 \sqcap T_2$ (that means that two locations of (element) type T_1 and T_2 can potentially evaluate to the same value.

Intuitively the meaning of formula $\psi(Mod^{@pre}, Mod^{@pre'}, Cap^{@pre'})$ is that all locations contained in Mod which have a reference type keep either their pre-state value or they hold a reference mentioned in the *captures* clause (which is evaluated in the pre-state). Again this PO requires a correct *assignable* clause to be sound.

Theorem 8 (Soundness and Completeness of *RespectsCap***)** *Let*

$$ct := (Pre, Post, Mod, Dep, Cap)$$

be a contract for op and Mod a correct assignable clause under the precondition

$$\alpha := Conj_{InvRealtime} \wedge Pre \wedge ValidCall_{op}^{SCJ}$$

The formula RespectsCap(op, ct) is valid if and only if Cap is a correct captures clause under the precondition α.

Proof 4 (Soundness and Completeness of *RespectsCap***)** *We first prove the soundness of PO RespectsCap by showing that an incorrect captures clause Cap for an operation op also entails invalidity of formula RespectsCap(op, ct) with ct := (Pre, Post, Mod, Dep, Cap) and then the completeness by showing that conversely invalidity of RespectsCap(op, ct) entails incorrectness of Cap with respect to op.*

Let us assume Cap is an incorrect captures clause. Then there are state s and r such that $s \models \alpha$ *and* $r' := \rho(p)(s)$ *with*

$$f^{r'}(a_1, \dots, a_n) \neq f^s(a_1, \dots, a_n) \tag{9.46}$$

$$f^{r'}(a_1, \dots, a_n) \neq val(\texttt{\textbf{null}}) \tag{9.47}$$

$$f^{r'}(a_1, \dots, a_n).\texttt{<c>}^s \neq val(FALSE) \tag{9.48}$$

$$f^{r'}(a_1, \dots, a_n) \neq g^s(b_1, \dots, b_m) \text{ f.a. } (g, (b_1, \dots, b_m)) \in val_s(Cap) \tag{9.49}$$

since the update $\mathcal{U} := \mathcal{D}@pre(Mod)||\mathcal{D}@pre'(Mod)||\mathcal{D}@pre'(Cap)$ *only assigns to virtual locations and thus does not affect the heap equations (9.46) through (9.49) still hold if we substitute* $r := \rho(p)(val_s(\mathcal{U})(s))$ *for* r'*.*

From equation (9.46) and Mod being a correct assignable *clause we deduce that*

$$(f, (a_1, \dots, a_n)) \in val_s(Mod) \tag{9.50}$$

and thus that there is a $ld \in Mod$ *with* $ld := (for\ \bar{x};\ if(\varphi)\ f(\bar{t}))$ *and a variable assignment β such that*

$$s, \beta \models \varphi \tag{9.51}$$

$$(f, (a_1, \dots, a_n)) = (f, (val_{s,\beta}(t_1), \dots, val_{s,\beta}(t_n))) \tag{9.52}$$

Consequently (since $r(f^{@pre}) = s(f)$ *and* $val_{r,\beta}(\bar{t}^{@pre}) = val_{s,\beta}(\bar{t})$*):*

$$r, \beta \overset{9.51}{\models} \varphi^{@pre} \tag{9.53}$$

$$val_{r,\beta}(f(\bar{t}^{@pre})) = f^r(a_1, \dots, a_n) \overset{(9.46)}{\neq} f^s(a_1, \dots, a_n) = val_{r,\beta}(f^{@pre}(\bar{t}^{@pre})) \tag{9.54}$$

$$val_{r,\beta}(f(\bar{t}^{@pre})) = f^r(a_1, \dots, a_n) \overset{(9.47)}{\neq} val(\texttt{\textbf{null}}) \tag{9.55}$$

$$val_{r,\beta}(f(\bar{t}^{@pre}).\texttt{<c>}^{@pre}) = f^r(a_1, \dots, a_n).\texttt{<c>}^s \overset{(9.48)}{\neq} val(FALSE) \tag{9.56}$$

$$r, \beta \overset{(9.49)}{\not\models} \bigvee_{\substack{c: c \in Cap^{@pre'} \wedge \\ ET(c), ET(m) \\ are\ compatible}} \exists \bar{x}_c; (\varphi_c \wedge f(\bar{t}^{@pre}) \doteq f_c^{@pre}(\bar{t}_c^{@pre})) \tag{9.57}$$

Therefore,

$$r \not\models \psi(Mod^{@pre}, Mod^{@pre'}, Cap^{@pre'}) \tag{9.58}$$

and together with the assumption $s \models \alpha$ *we get*

$$s \not\models RespectsCap(op, ct) \tag{9.59}$$

Let us now assume $RespectsCap(op, ct)$ *is invalid. Then there is a Kripke structure* $\mathcal{K} := (\mathcal{M}, \mathcal{S}, *, \rho)$ *and states* s' *and* $r := \rho(p)(val_{s'}(\mathcal{U})(s'))$ *with*

$$\mathcal{U} := \mathcal{D}@pre(Mod)||\mathcal{D}@pre'(Mod)||\mathcal{D}@pre'(Cap)$$

and $s', r \in \mathcal{S}$ *such that*

$$\begin{aligned} s' &\models \alpha && (9.60)\\ t &\not\models \psi(Mod^{@pre}, Mod^{@pre'}, Cap^{@pre'}) && (9.61) \end{aligned}$$

We define $s := val_{s'}(\mathcal{U})(s')$. *Then* $r := \rho(p)(s)$ *and due to equation (9.60) and* $\mathcal{U}$ *only updating virtual locations:*

$$\begin{aligned} s &\models \alpha && (9.62)\\ val_s(Cap) &= val_{s'}(Cap) && (9.63)\\ val_s(Mod) &= val_{s'}(Mod) && (9.64) \end{aligned}$$

Together with equation 9.61 this implies that there is a $(f, (a_1, \ldots, a_n)) \in val_s(Mod)$ *such that*

$$\begin{aligned} f^r(a_1, \ldots, a_n) &\neq f^s(a_1, \ldots, a_n) && (9.65)\\ f^r(a_1, \ldots, a_n) &\neq val(\texttt{\textbf{null}}) && (9.66)\\ f^r(a_1, \ldots, a_n).\texttt{<c>}^s &= val(TRUE) && (9.67)\\ f^r(a_1, \ldots, a_n) &= g^s(b_1, \ldots, b_m)\ f.a.\ (g, (b_1, \ldots, b_m)) \in val_s(Cap) && (9.68) \end{aligned}$$

Equations (9.65) through (9.68) indicate that Cap is an incorrect captures clause according to definition 9.2. □

Example 9.11 (*RespectsCap* **for a setter method)** *Let us consider the following JML-specified method implemented in a class C:*

— Java + JML —

```
/*@ public normal_behavior
  @  assignable this.attr;
  @  depends this, a;
  @  captures a;
  @*/
public void setAttr(Object a){
  this.attr = a;
}
```

— Java + JML —

The JML method spec is equivalent to the contract

$$ct := (true, true, \{self.attr\}, \{self, a\}, \{a\})$$

Thus the proof obligation RespectsCap$(setAttr, ct)$ is given by:

$$\alpha \rightarrow \left\{\begin{array}{l} self^{@pre} := self|| \\ for\,C\ x;\ if(true)\ attr^{@pre}(x) := x.attr|| \\ a^{@pre} := a \end{array}\right\} [Prg_{setAttr}()]\psi_{setAttr}$$

where

$$\psi_{setAttr} := \begin{array}{l} self^{@pre}.attr \doteq attr^{@pre}(self^{@pre}) \vee \\ self^{@pre}.attr \doteq \texttt{\textbf{null}} \vee \\ self^{@pre}.attr.\texttt{<c>}^{@pre} \doteq FALSE \vee \\ self^{@pre}.attr \doteq a^{@pre} \end{array}$$

and α is defined as before.

9.3 Alternative Approach

The Java DL semantics for *assignable* and *depends* clauses, though sensible for the verification of sequential programs, is to liberal for proving non-interference. Even a method with an empty *assignable* clause is allowed to temporarily modify data which can cause interference. This can be illustrated by method `doNothing` in Section 9.2. Let us assumed `doNothing` is concurrently executed by two threads: If the execution of both threads is interlaced like this

— Java + JML —

```
int i = o.a++;     //executed by Thread 1
int i = o.a++;     //executed by Thread 2
o.a = i;           //executed by Thread 1
o.a = i;           //executed by Thread 2
```

— Java + JML —

the field `o.a` has been increased by `1` after `doNothing` was executed by the two threads despite its empty *assignable* and *depends* clause.

This example illustrates that for deciding whether the execution of a method can cause interference with other threads *assignable* and *depends* clauses as interpreted in Java DL are not sufficient. Before tackling this issue we have to clarify what the interesting properties (with respect to the ensuring of non-interference) of a program are. As pointed out in Lemma 9.1, a method `m` can show no interference (neither actively by modifying data other threads read, or passively be reading data that is concurrently modified by another thread) if no location read or written by `m`, if executed by a thread different from the initial one, resides in *immortal* or *mission* memory[6].

Forbidding read access to data in *immortal/mission* memory is a strong restriction. Non-interference is ensured as well in KeYSCJ* if no locations in *immortal* or *mission* memory

[6] This should not be confused with where the value a location refers to is allocated. The memory area of a location $o.a$, for instance, denotes the memory area in which o resides since the field a (containing either a reference or a primitive value) is part of o, not the memory area of the object referenced by a (in case a has a reference type).

are modified by *any* thread, except the initial one, so this might be the second "interesting" property to check on the method level.

After we have identified what to verify (namely that no write access to *immortal* or *mission* memory occurs) the question is how to verify it: Writing proof obligations in the manner shown in Section 9.2.1 making statements about the state change performed by a program won't succeed since we must also account for intermediate, temporary changes. An alternative could be to extend the calculus in a way to reflect additional constraints on the KeYSCJ* profile in dependence of the property we aim to verify. Technically this can be realized in the KeY systems be means of alternative rule sets[7].

We discuss briefly the extensions needed to ensure the following property to which we will refer to as the *WriteRestricted* KeYSCJ* criterion:

> For any write access to a location l executed by a thread t, l must either reside in a scope s with:
>
> $$t.getMemoryArea().\texttt{stack} \preceq s.\texttt{stack}$$
>
> or l must be an instance field of $t.getMemoryArea()$ (we need to make this exception to permit a thread to enter its initial scope, and thus modify some of its attributes, when it starts executing).

The currently executing `RealtimeThread` can be obtained by the RTSJ API method:

`RealtimeThread.currentRealtimeThread()`

The required changes to ensure this property solely affect the calculus rules executing assignments to attributes and array slots. Concretely this means that for each assignment to a (reference as well as primitive type) field $o.a$ we have to ensure that:

$$\begin{array}{l} RealtimeThread.currentRealtimeThread().getMemoryArea().\texttt{stack} \preceq o.\texttt{<ma>}.\texttt{stack} \vee \\ RealtimeThread.currentRealtimeThread().getMemoryArea() \doteq o \end{array}$$

This leads to the rule $\mathsf{pAttrWrite}'$ for assignments to primitive type attributes:

$$\mathsf{pAttrWrite}' \quad \frac{\begin{array}{l} \Gamma \Rightarrow \{\mathcal{U}\}(ct.gma().\texttt{stack} \preceq o.\texttt{<ma>}.\texttt{stack} \vee ct.gma() \doteq o),\ \Delta \\ \Gamma,\ \{\mathcal{U}\}(ct.gma().\texttt{stack} \preceq o.\texttt{<ma>}.\texttt{stack} \vee ct.gma() \doteq o) \Rightarrow \{\mathcal{U}\}\{o.a := v\}\langle \texttt{p} \rangle \varphi,\ \Delta \end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle \texttt{o.a=v; p} \rangle \varphi,\ \Delta}$$

and to the rule $\mathsf{rAttrWrite}'$ for assignments to reference type attributes:

$$\mathsf{rAttrWrite}' \quad \frac{\begin{array}{l} \Gamma,\ \{\mathcal{U}\}(o \not\doteq \textbf{null} \wedge \psi) \Rightarrow \{\mathcal{U}\}(ct.gma().\texttt{stack} \preceq o.\texttt{<ma>}.\texttt{stack} \vee ct.gma() \doteq o),\ \Delta \\ \Gamma,\ \{\mathcal{U}\}(o \not\doteq \textbf{null} \wedge \psi \wedge (ct.gma().\texttt{stack} \preceq o.\texttt{<ma>}.\texttt{stack} \vee ct.gma() \doteq o)) \Rightarrow \\ \qquad \{\mathcal{U}\}\{o.a := v\}\langle \texttt{p} \rangle \varphi,\ \Delta \\ \Gamma,\ \{\mathcal{U}\}o \doteq \textbf{null} \Rightarrow \{\mathcal{U}\}\langle \texttt{NPE; p} \rangle \varphi,\ \Delta \\ \Gamma,\ \{\mathcal{U}\}(o \not\doteq \textbf{null} \wedge \neg\psi) \Rightarrow \{\mathcal{U}\}\langle \texttt{IAE; p} \rangle \varphi,\ \Delta \end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle \texttt{o.a=v; p} \rangle \varphi,\ \Delta}$$

[7] KeY already possesses alternatively selectable rule sets reflecting, for instance, different integer semantics (mathematical (unbounded) semantics and Java (bounded) semantics).

where $ct := \mathit{RealtimeThread.currentRealtimeThread}()$, $\mathit{gma}() := \mathit{getMemoryArea}()$ and $\psi := v \doteq \texttt{null} \vee v.\texttt{<ma>}\,\texttt{.stack} \preceq o.\texttt{<ma>}\,\texttt{.stack}$. Write access to array slots can be treated analogously.

When assigning to a static attribute, we have to ensure that the executing thread resides in *immortal* memory. Again this holds for primitive as well as reference type attributes. The rule sRAttrWrite′ handling assignments to reference type static fields can be defined as:

$$\mathsf{sRAttrWrite'}\ \frac{\begin{array}{l}\Gamma,\ \{\mathcal{U}\}(\mathsf{im}(v.\texttt{<ma>}\,\texttt{.stack})) \Rightarrow \mathsf{im}(ct.\mathit{gma}().\texttt{stack}),\ \Delta \\ \Gamma,\ \{\mathcal{U}\}((\mathsf{im}(v.\texttt{<ma>}\,\texttt{.stack}) \vee v \doteq \texttt{null}) \wedge \mathsf{im}(ct.\mathit{gma}().\texttt{stack})) \Rightarrow \\ \quad \{\mathcal{U}\}\{sa := v\}\langle \texttt{p} \rangle\, \varphi,\ \Delta \\ \Gamma,\ \{\mathcal{U}\}(\neg\, \mathsf{im}(v.\texttt{<ma>}\,\texttt{.stack}) v \wedge \not\doteq \texttt{null}) \Rightarrow \{\mathcal{U}\}\langle \texttt{IAE;}\ \texttt{p} \rangle\, \varphi,\ \Delta\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle \texttt{sa=v;}\ \texttt{p} \rangle\, \varphi,\ \Delta}$$

The corresponding rule for primitive type static fields is obtained analogously.

Example 9.12 (*doNothing* Revisited) *Let us again consider the method* `doNothing` *presented in Section 9.2. With regard to the* WriteRestricted *criterion the original specification is no longer correct since the operations* `o.a++` *and* `o.a = i` *necessitate that*

$$\mathit{RealtimeThread.currentRealtimeThread}().\mathit{getMemoryArea}().\texttt{stack} \preceq o.\texttt{<ma>}\,\texttt{.stack}$$

which is checked by rule pAttrWrite′. *Thus we have to adapt the original specification accordingly:*

— JAVA + JML —

```
/*@ requires o!=null &&
  @   \outerScope(RealtimeThread.currentRealtimeThread().getMemoryArea(),
  @               \memoryArea(o));
  @ ...
  @*/
public void doNothing(MyObject o){
  int i = o.a++;
  o.a = i;
}
```

— JAVA + JML —

Specifications that are correct with regard to the WriteRestricted *criterion for KeYSCJ* are, however, also correct in its unrestricted semantics which can thus be considered a* refinement *of the semantics including the* WriteRestricted *criterion.*

9.3.1 Handling of Threads

Since we ensured that threads complying with the KeYSCJ* profile show no interference, we may treat them as if executed sequentially. In particular the symbolic execution of the `start()` method of a `javax.realtime.RealtimeThread` is handled like for any "normal" JAVA method sequentially. For realizing this approach one technical issue remains: we have to keep track of the currently executing thread.

This can be solved largely analogously to the handling of the `<cma>` pointer (see Section 6.2.1) referencing the currently active memory area and it is therefore not discussed in too much detail here. In correspondence to the approach described in Section 6.2.1 we introduce the pointer `<ct>` referring to the currently active thread and include a reference to this current thread in the `method-frame` statement:

Java

```
method-frame(result->retvar,
             source=T,
             this=self,
             <cma>=mem,
             <ct>=thread) : {body}
```

Java

Since except for the method `start()@javax.realtime.RealtimeThread` (where the receiver object of the method call starts executing) the executing thread does not change when a method call occurs, we only have to pay special regard to this method:

$$\text{expandStart}\ \frac{\Gamma \Rightarrow \langle \pi\ \texttt{method-frame(}\ \begin{array}{l}\texttt{result->lhs,}\\ \texttt{source=MemoryArea,}\\ \textbf{this}\texttt{=se}\\ \texttt{<cma>=cma}\\ \texttt{<ct>=se)} : \{\texttt{body}\}\ \omega\rangle\phi,\ \Delta\end{array}}{\Gamma \Rightarrow \langle \pi\ \texttt{lhs=se.start()@(RealtimeThread);}\ \omega\rangle\ \phi,\ \Delta}$$

where *cma* denotes the current scope as determined by π. The method body of `start()` is given by:

Java

```
getMemoryArea().enter(this);
```

Java

When expanding method body statements of other methods, we just set `<ct>` to the value `<ct>` refers to in the enclosing method frame.

Having defined the `<ct>` pointer we can now also provide an implementation for the method `RealtimeThread.currentRealtimeThread()` which is simply

Java

```
public static RealtimeThread currentRealtimeThread(){
  return <ct>;
}
```

Java

The Default Thread

Initially when the value of `<ct>` is not explicitly determined by a method frame `<ct>` evaluates to the program variable $defaultThread$. This program variable is used in a very similar way as $defaultMemoryArea$ (see Section 6.2.1) and $self$. It can be used in method contracts to specify the thread executing the specified method and (comparable $defaultMemoryArea$ and $self$) needs to be instantiated when the contract is applied with the thread `<ct>` refers to at the point of application of the contract.

Restrictions

Asynchronous Event Handler Instances of class `javax.realtime.AsyncEventHandler` are, like `RealtimeThreads`, `Schedulable` objects. Their execution is not synchronously started by the application but triggered by an event they are associated with. They are then either executed by a dedicated thread the event handler is bound to (in case it is a `BoundAsyncEventHandler`) or by an arbitrary already existing server thread.

We do not model events that trigger the execution of `AsyncEventHandlers` at arbitrary times. This is, however, no major restriction since if it is ensured that `AsyncEventHandlers` comply with the KeYSCJ* profile they are known to show no interference with other threads and can thus safely be executed at arbitrary times.

Example 9.13 *When symbolically executing the following method, the* `run` *method of the newly created* `AsyncEventHandler` *is not symbolically executed:*

— *Java* —

```java
public void createAsyncEventHandler(Runnable logic){
  new AsyncEventHandler(logic);
}
```

— *Java* —

Assumptions on the Application The described approach assumes compliance of the regarded application with the properties of the KeYSCJ* profile described in Section 9.1.1 but does not ensure it. Therefore, these properties must be ensured differently, for example by syntactic checks (which are sufficient for ensuring that `getPortal` and `setPortal` are not used) or by reviews. Writing programs that trustably comply with the KeYSCJ* profile in an obvious way should, however, be relatively simple.

Part III

Modular Verification of WCMU Contracts

10 The Necessity for Correct *Worst Case Memory Usage* Estimations

Estimating the worst case memory usage (WCMU) of a Java application is essential for giving performance or safety guarantees. This becomes even more relevant in the context of real-time and embedded applications where the amount of memory available is inherently small and software failures caused by memory shortage are not acceptable. In the context of real-time Java [Bollella and Gosling, 2000] estimating the WCMU is especially relevant with regard to the concept of scoped memory [Beebee and Rinard, 2001], which allows defining memory areas of a fixed size (memory scopes) that are not subject to garbage collection. Since unreferenced objects in scopes are not recycled by the garbage collector, which can easily give rise to memory leaks, it would be desirable to have a means of verifying that the heap space allocated by an application does not exceed a certain upper bound, for instance a given scope size.

Another field of application [Giambiagi and Schneider, 2005] for WCMU analysis techniques are smart cards which usually possess only several KB of RAM memory. This application scenario is particularly relevant for this work since the KeY tool which has been used for a prototypical implementation of the presented technique is targeted on the verification of programs written in Java Card, a Java dialect for smart cards.

Even though the usefulness and necessity for a methodology for ensuring WCMU constraints is evident [Jacobs et al., 2007], only few theoretical works [Krone et al., 2001, Atkey, 2006, Hayes and Utting, 2001, Hunt et al., 2006, Barthe et al., 2005] in this area exist. In practice, due to the lack of static analysis tools in this field, WCMU is often validated experimentally (by measuring the memory usage during runtime). However, this can, like testing in general, give no guarantees on the correctness of the tested WCMU estimations.

In the following we elaborate on the specification and verification of WCMU constraints. Existing WCMU specification means in JML are described and their shortcomings are pointed out. We propose extensions to JML facilitating modularity of the specification and add to overcoming the identified shortcomings. We then show how the extension we made are reflected in Java DL and handled in the calculus. Here we put a focus on modular verification of WCMU contracts and the handling of loops. Finally we sketch how MIB memory models, in particular the one featured by PERC Pico, affect the presented approach.

11 JML Memory Performance Specifications

11.1 Existing JML Specifications and their Shortcomings

Besides a variety of constructs aiming at the description of the functional behavior of Java programs, JML provides means for specifying performance properties such as worst case execution time and heap memory consumption. The worst case heap memory allocation of a method under a certain precondition `PRE` is specified as part of a JML method specification using the **`working_space`** clause:

— Java + JML —

```
/*@ public normal_behavior
  @  requires PRE;
  @  ...
  @  working_space S;
  @*/
 public void foo(){...
```

— Java + JML —

`S` is a JML expression of type **`long`** that defines an upper bound on the size of heap space allocated by `foo()` when invoked in a state satisfying `PRE`. `S` is evaluated in the post-state of `foo()` and allows (like other JML clauses evaluated in the post-state) access to the pre-state by JML's **`\old`** construct. As already identified in [Atkey, 2006], this restriction to two program states can be seen as a severe shortcoming of JML's memory consumption specs, on which this work will elaborate in the following.

The JML reference manual [Leavens et al., 2007] does not state clearly if and how garbage collection (GC) is taken into account by a **`working_space`** clause. One could think of different approaches here:

1. The effect of GC is ignored. Thus every object allocation performed by the method contributes to its memory usage and has to be taken into account by the **`working_space`** clause irrespective of the lifetime of the object. The **`working_space`** clause consequently specifies the worst case amount of heap space additionally (compared to the pre-state) consumed in the post-state of the regarded method.

2. The effect of GC is taken into account and the **`working_space`** clause specifies

   a) the additional amount of memory still consumed (an not reclaimable by GC) in the worst case in the post-state of the regarded method or

   b) the worst case amount of memory additionally consumed (and not reclaimable by GC) in any program state during execution of the regarded method.

As our focus is on RTSJ, SCJ and Java Card applications, which are not subject to garbage collection anyway, we adopt the first approach here.

The space allocated to an object `o` (only to the object `o` itself not including objects referenced by `o`'s attributes) can be obtained by `\space(o)`.
The JML function `\working_space` (not to be confused with the clause `working_space` used above) describes

> the maximum specified amount of heap space, in bytes, used by the method call or explicit constructor invocation expression that is its argument. (JML Reference Manual [Leavens et al., 2007])

Example 11.1 *The amount of memory allocated by* `m` *according to its JML specification shown below can be referred to in other JML specifications by means of the* `\working_space` *function.*

— Java + JML —

```
public myClass{ ...

 /*@ public normal_behavior
   @  requires a>0;
   @  working_space 0;
   @ also public normal_behavior
   @  requires a<=0;
   @  working_space 8;
   @*/
 public static Object m(int a){ ...
```

— Java + JML —

The expression `\working_space(myClass.m(3))`*, for instance, could be replaced by* `0` *since there is only one specification case, namely the first one in the above code, whose precondition (*`a>0`*) is true if* `m` *is called with the argument* `3`*.*

Considering, however, the following piece of code:

— Java + JML —

```
 /*@ public normal_behavior
   @  ...
   @  ensures \result!=0;
   @  working_space
   @     \working_space(myClass.m(\result));
   @*/
 public int m2(){...}
```

— Java + JML —

both preconditions(`a>0` *and* `a<=0`*) can hold in the state the expression*

— Java + JML —

```
 \working_space(myClass.m(\result))
```

— Java + JML —

is evaluated in as we only know that `\result!=0` *holds in this state. Thus, all we can assume without additional information is that this expression is smaller or equal to the maximum of the specified working space clauses of both specification cases, which is* `8`*.*

As example (11.1) illustrated, an expression `\working_space(m())` used in the working space clause of a specification case of a method `m2` denotes the worst case memory consumption of `m` as derivable from those of `m`'s specification cases, whose precondition can potentially be true in the state `\working_space(m())` is evaluated in (either the pre or post-state of `m2`, depending on whether the expression `\working_space(m())` occurs inside an `\old` expression or not). This means, in other words, we need to consider all specification cases of `m` possessing a precondition which is not contradictory to the precondition (if `\working_space(m())` is evaluated in the pre-state of `m2`) of the specification case `\working_space(m())` occurs in or to its post condition (if `\working_space(m())` is evaluated in the post-state) respectively.

One could argue that a more fine grained `\working_space` function permitting to refer to the working space of a single specification case, would be desirable to be able to write more precise working space specifications.

Besides the fact that these restrictions (the lack of granularity and the restriction to the pre- and post-state of the specified method) can be considered inconvenient, they could easily give rise to specification bugs as the following example shows:

Java + JML

```
public static SomeClass _instance;

/*@ public normal_behavior
  @  working_space 0;
  @  assignable _instance;
  @  ensures _instance==null;
  @*/
public static SomeClass clear(){
    SomeClass old = _instance;
    _instance = null;
    return old;
}

/*@ public normal_behavior
  @  requires _instance==null;
  @  assignable _instance;
  @  working_space \space(new SomeClass());
  @ also public normal_behavior
  @  requires _instance!=null;
  @  assignable \nothing;
  @  working_space 0;
  @*/
public static SomeClass getInstance(){
    if(_instance==null) _instance = new SomeClass();
    return _instance;
}
```

```
/*@  requires _instance!=null;
  @  working_space \working_space(clear()) +
  @            \working_space(getInstance());
  @*/
public SomeClass freshInstance(){
    clear();
    return getInstance();
}
```

Java + JML

The above specification of `freshInstance()` is incorrect since in the state `getInstance()` is called

$$\mathit{\_instance} \doteq \mathit{null}$$

holds while in the post-state in which we evaluate `\working_space(getInstance())`

$$\mathit{\_instance} \not\doteq \mathit{null}$$

holds leading to a specified working space of `0` (instead of `\space(new SomeClass())`) for this case. Writing `\old(\working_space(getInstance()))` does not help either since in the pre-state `_instance` is also required not to be **`null`**. This example illustrates that

- it is not possible with the JML semantics as it is to specify the working space of method `freshInstance()` relative to the working space of `getInstance()` and `clear()` since one does not have access to the state in which `getInstance()` is called in the code and
- that the semantics of the `\working_space` expression makes JML vulnerable to specification bugs since the programmer could be tempted to use method calls occurring in the specified method's body by just copying and pasting them into the **`working_space`** clause as has happened in the above specification.

11.2 Enhanced JML Heap Memory Specifications

This work proposes an extension and modification of the JML construct `\working_space` addressing both drawbacks of the current concept: the restriction to the pre- and post-state of the specified method and the lack of granularity. To distinguish the present JML specification from our proposal we will refer to the latter as the KeYJML spec.

The syntax and semantics of KeYJML memory specifications differs in certain respects from the original JML specification. Concretely these differences concern

- the applied integer semantics
- the state in which working space clauses are interpreted
- a modification of the `\working_space` function and
- an extension of JML's loop specifications.

11.2.1 The Applied Integer Semantics

Since KeY supports mathematical (unbounded) integers, `\space` and `\working_space` expressions as well as the expressions contained in `working_space` clauses do not have the Java type `long` but are treated by KeY as mathematical integers. This makes verification tasks a bit easier since modulo arithmetics is no longer required for evaluating working space clauses themselves[1] and prevents subtle (specification) bugs that can arise if integer overflows are not taken into account.

11.2.2 Interpretation of Working Space Clauses in the Pre-State

In contrast to JML, `working_space` clauses in KeYJML are interpreted in the pre-state. This is motivated by the rationale that in scenarios in which performance and especially memory consumption specifications are of interest, namely for real-time and embedded systems with a possibly very limited amount of physical memory or scoped memory in a certain scope (as in RTSJ), it is desirable to estimate the memory allocation of a method based on the information available at the point it is called (i.e., its pre-state). However, the approach presented in this paper does not depend on this decision and keeping JML's original semantics would only require minor modifications in some of the calculus rules shown in Section 13.

11.2.3 Rigid Working Space Functions with Explicit Preconditions

The `\working_space` function undergoes the most distinct changes syntactically and semantically compared to the original JML definition. Beside supporting the original JML working space function KeYJML incorporates a new working space function that is written as

— JML —

```
\working_space(m, pre)
```

— JML —

where `m` is a method signature and `pre` is a `boolean` expression. `\working_space(m, pre)` then denotes the maximum of the specified amount of heap space consumed by `m` if invoked in a state satisfying `pre`. This means that we have to take the maximum over all working space clauses having a precondition which is not contradictory to `pre`. If `pre` is chosen carefully this is only the case for exactly one contract. The second argument `pre` can be thought of as quoted meaning it is not evaluated in the state the containing expression `\working_space(m, pre)` occurs in, thus, making the entire expression `\working_space(m, pre)` not state dependent (i.e., rigid). The expression

— JML —

```
\working_space(SomeClass.m(int a1, int a2), a1<a2)
```

— JML —

for instance denotes the maximum amount of heap space method `SomeClass.m` can consume according to its specification if invoked in a state in which the first of `m`'s arguments is smaller than the second one (`a1<a2`). This alternative suggestion of defining the `\working_space` function helps to overcome the drawbacks described in Section 11.1 since

[1] Of course we still need modulo arithmetics for reasoning over programs containing arithmetic operations on integers.

- by making the pre-state (of the method) and thus the relevant specifications explicitly selectable, it supports a finer level of granularity than the original JML variant of the `\working_space` function and
- reduces the risk of "copy and paste" related specification bugs as demonstrated in Section 11.1.

The working space of `freshInstance` we were unable to specify conveniently with standard JML in the preceding example can be expressed as

— JML —

```
\working_space(clear()) +
\working_space(getInstance(), _instance==null)
```

— JML —

11.2.4 Loop Working Space Specifications

Since JML lacks features for specifying the memory consumption of loops, we propose a working space clause applicable to loops which specifies the maximum amount of heap memory allocated by any single loop iteration not terminating with an exception or by a **break** statement. We denote this clause **wssi** (shorthand for working space single iteration) in the following to distinguish it from method level working space clauses. The **wssi** clause is evaluated in the state before the first iteration of the loop. Its value in this state is therefore required to be an upper bound for the WCMU of any subsequent loop iteration terminating normally.

An upper bound of the accumulated amount of memory consumed by the loop in all its normally terminating iterations is then given by $dec * w$, where dec and w are the pre-state values of the expressions specified by the **decreasing** and the **wssi** clause. The **decreasing** clause specifies a value that is (i) strictly decreasing in every iteration of the loop and (ii) always greater 0. Thus dec constitutes an upper bound for the number of loop iterations. The specification of `initArr` shown below illustrates the usage of the **wssi** clause which is, as method-level working space clauses, preceded by the **working_space** keyword:

— Java + JML —

```
/*@ public behavior
  @  requires a!=null;
  @  working_space s1 > s2 ? s1 : s2;
  @*/
 public void initArr(Object[] a){
     int i=0;
     /*@ loop_invariant i>=0;
       @ assignable a[*], i;
       @ decreasing a.length-i;
       @ working_space \space(new Object());
       @*/
     while(i<a.length){
         a[i++] = new Object();
     }
}
```

Java + JML

where s1 is an abbreviation for

JML

```
\working_space(new ArrayStoreException(), true)+\space(new Object())
```

JML

and s2 for

JML

```
a.length*\space(new Object())
```

JML

In each normally terminating iteration of the above **while** loop an object of type `Object` is created. Thus, `\space(new Object())` is a correct **wssi** clause. In case the runtime type of `a` is a strict subtype of `Object[]`, an `ArrayStoreException` is raised and the memory consumption of the loop body would be the working space of the constructor call `new ArrayStoreException()` which includes the space occupied by the newly created `ArrayStoreException` itself. Since, if this happens, the loop body does not terminate normally, this case needs not be taken into account according to our definition of the semantics of **wssi**. However, it has to be taken into account when specifying `initArr`'s working space. This example also clarifies the rationale behind defining the **wssi** clause only for normally terminating iterations of the loop: The working space of `new ArrayStoreException()` (several hundreds of bytes, depending on the created stack trace) is significantly larger then the space occupied by a newly created object of type `Object` (8 bytes for the JVM characteristics we exemplarily consider here). If w were also required to be an upper bound for the heap space consumed by an abruptly terminating iteration the value $dec * w$ would be of no real significance for the worst case memory consumption estimation of the loop, since:

- If the loop raises no exception w is by an order of magnitudes larger then the space actually consumed by each iteration (`\space(new Object())`) which also applies to the worst case estimation $dec * w$ for the memory consumption of the entire loop.
- If the loop raises an exception, it is only executed once and our worst case estimation $dec * w$ would be wrong by factor dec.

By restricting **wssi** the way it is done we get a precise worst case estimation for the first of the above two cases. This information can also be used by the Java DL calculus for determining a correct upper bound for the memory consumption of a loop terminating abruptly in an arbitrary iterationas shown in Section 13 .

Dependent wssi Clauses

In method `initArr` the value (`\space(new Object())`) specified by the **wssi** clause exactly matches the (constant) memory consumption of each normally terminating loop iteration. In cases the memory consumption of a loop is not constant but varies significantly over different iterations the presented approach could, however, lead to imprecise WCMU estimations. This can be exemplified by the following piece of code:

Java + JML

```
/*@ public behavior
  @  requires a!=null && typeof(a)==Object[];
  @  working_space (\sum int i; 0<=i && i<a.length && a[i]==null;
  @     \space(new Object()));
  @*/
 public void completeArr(Object[] a){
     int i=0;
     /*@ loop_invariant i>=0;
       @ assignable a[*], i;
       @ decreasing a.length-i;
       @ working_space \space(new Object());
       @*/
     while(i<a.length){
         if(a[i]==null) a[i] = new Object();
         i++;
     }
 }
```

Java + JML

The above method level *working space* clause is obviously correct. However, we cannot deduce its correctness from the WCMU estimation we obtain for the loop which is

JML

```
\space(new Object())*a.length
```

JML

based on the **`wssi`** and the **`decreasing`** clause since it is a to rough overapproximation of the loop's actual memory consumption which matches the WCMU WS of `completeArray` as specified by its method-level *working space* clause[2]:

$$\sum_{\substack{i:i\in\{0,\ldots,a.length-1\}\wedge\\ \backslash\mathbf{old}(a[i])=\mathbf{null}}} \texttt{\textbackslash}\mathtt{\textbf{space}(\textbf{new}\ Object())}$$

We can overcome this shortcoming by using a loop-level *working space* clause that can vary over different loop iterations. For the above loop the memory consumption of a single iteration is:

$$w(i) := \begin{cases} \texttt{\textbackslash}\mathtt{\textbf{space}(\textbf{new}\ Object())} & \text{iff. } \texttt{\textbackslash}\mathtt{\textbf{old}(a[i])} = \mathtt{\textbf{null}} \\ 0 & \text{iff. } \texttt{\textbackslash}\mathtt{\textbf{old}(a[i])} \neq \mathtt{\textbf{null}} \end{cases}$$

The memory consumption of the entire loop could then be determined by

$$\sum_{i=0}^{a.length-1} w(i)$$

[2] To avoid too much notational overhead mathematical and JML notations are sometimes mixed in this motivating example. A formal semantics of the *dependent* **`wssi`** *clause* is given by rule loopInvTotalVarWS in Section 13.2.

which matches WS. This illustrates that we could achieve improved preciseness by a loop level *working space* that is variable over different loop iterations.

The above considerations lead us to the definition of a loop-level *working space* clause specifying the memory consumption of the i$^{\text{th}}$ iteration of the loop:

```
— JML ——————————————————————————————————————
 working_space(i) w;
—————————————————————————————————————— JML —
```

where

- i is a variable of type long bound in w.
- w may not depend on any locations contained in the loops *assignable* clause.

This variable **`wssi`** (we call it **`vwssi`** in the following) clause specifies the WCMU of the i$^{\text{th}}$ iteration of the loop. At the beginning of an arbitrary iteration the maximum amount of memory consumed so far is then given by:

$$\sum_{i=0}^{dec_0-dec_i-1} w$$

where dec_i stands for the value of the loop variant (specified by the decreasing clause) before the (i+1)$^{\text{th}}$ iteration of the loop.

The WCMU of the loop occurring in method `completeArray` could, for instance, be specified by the clause

```
— JML ——————————————————————————————————————
 working_space(i) \old(a[i])==null ? \space(new Object()) : 0;
—————————————————————————————————————— JML —
```

which precisely reflects the WCMU of the (i+1)$^{\text{th}}$ iteration of the loop.

11.2.5 Assumptions on the Java Virtual Machine

The approach presented in this work is independent of characteristics, such as the memory overhead needed to store an object and alignment issues, of the Java Virtual Machine (JVM) the regarded code runs on. Nevertheless for determining the concrete memory consumption of an application on a specific target platform we need to know certain characteristics of the target platform (in the calculus presented in Section 13 we therefore distinguish platform independent rules and platform specific rules). For the calculus rules and examples presented in the following we will assume the JVM characteristics of the *Sun J2SE 1.4.2 32bit Client VM* running on the Linux operating system. This entails that we can provide concrete values for the space occupied by objects and arrays (see Section 13), in case their dimension is known.

In particular the JVM characteristics we assume are:

- The space $as_{e,l}$ required for a one-dimensional array of length l with each of its entries occupying e bytes is:

$$as_{e,l} = min\{a|a \geq 12 + e * l \wedge a\ mod\ 8 \equiv 0\}$$

- The space space_T in bytes occupied by an object of type T is:

  $$\mathit{space}_T := \mathit{min}\{a | a \geq 8 + s \wedge a \; \mathit{mod} \; 8 \equiv 0\}$$

  where s is the space occupied by the fields of the object.

11.2.6 Mapping JML Expressions to Java DL

For making JML expressions utilizable within KeY, it is necessary to compile them to Java DL.

As before, T^{DL}_{JML} denotes the mapping from JML expressions to Java DL terms and formulas. For the JML functions `\working_space` and `\space` we define T^{DL}_{JML} as:

- $T^{DL}_{JML}($`\working_space(C.m(T1 p1,...,Tn pn), pre)`$) := \mathit{ws}^{r}_{C.m(\bar{p}),pre}$,
  where $\bar{p} := T_1\, p_1, \ldots, T_n\, p_n$ and $\mathit{ws}^{r}_{C.m(\bar{p}),pre}$ (with $p_i := T^{DL}_{JML}($`pi`$)$ and $T_i := T^{DL}_{JML}($`Ti`$)$) is a rigid term. This means in particular that *pre* can be thought of as quoted since it is not evaluated in the state $\mathit{ws}^{r}_{C.m(\bar{a}),pre}$ occurs in.

- $T^{DL}_{JML}($`\working_space(self.m(T1 p1,...,Tn pn), pre)`$) := \mathit{ws}^{r}_{self.m(\bar{p}),pre}$

- $T^{DL}_{JML}($`\working_space(C.m(a1,...,an))`$) := \{\mathcal{U}\}\mathit{ws}^{nr}_{C.m}[\mathit{Dep}](\bar{p})$,
  where

  - *Dep* denotes *m*'s *depends* clause.
  - $\bar{p} := p_1, \ldots, p_n$ are the parameters of m used in its declaration (and thus also its contract).
  - $\mathit{ws}^{nr}_{C.m}$ is a location dependent function symbol.
  - The update $\{\mathcal{U}\} := \{p_1 := a_1 || \ldots || p_n := a_n\}$ "binds" the formal parameters $\bar{p}$ to the actual arguments $\bar{a}$ (with $a_i := T^{DL}_{JML}($`ai`$)$).

- $T^{DL}_{JML}($`\working_space(o.m(a1,...,an))`$) := \{\mathcal{U}\}\mathit{ws}^{nr}_{m}[\mathit{Dep}](\bar{p})$
  with $\bar{p} := \mathit{self}, p_1, \ldots, p_n$ and $\{\mathcal{U}\} := \{\mathit{self} := o || p_1 := a_1 || \ldots || p_n := a_n\}$.
  In the following when discussing flexible (location dependent) working space terms we may omit the *depends* clause *Dep* in cases it is not explicitly required for our considerations and just write $\mathit{ws}^{nr}_{m}(\bar{a})$ instead of $\mathit{ws}^{nr}_{m}[\mathit{Dep}](\bar{a})$.

- $T^{DL}_{JML}($`\space(new T())`$) := i$,
  with the integer literal i being the amount of heap space an object of type T occupies.

- For representing `\space` expressions whose arguments have an array type we introduce a rigid function symbol space^{arr}, where $\mathit{space}^{arr}(s, l)$ denotes the space occupied by a one-dimensional array of length l whose entries (for primitive typed entries) or entry references (in case of reference typed entries) respectively have size s.

  $T^{DL}_{JML}($`\space(new T[d1]...[dn][]...[])`$) :=$
  $\mathit{space}^{arr}(4, T^{DL}_{JML}(d1)) + T^{DL}_{JML}($`d1`$) * T^{DL}_{JML}($`\space(new T[d2]...[dn][]...[])`$)$
  $T^{DL}_{JML}($`\space(new T[d][]...[])`$) := \mathit{space}^{arr}(4, T^{DL}_{JML}($`d`$))$
  $T^{DL}_{JML}($`\space(new T[d])`$) := \mathit{space}^{arr}(s, T^{DL}_{JML}($`d`$))$

where

$$s := \begin{cases} 1 & \text{iff. } T \in \{byte, boolean\} \\ 2 & \text{iff. } T \in \{short, char\} \\ 4 & \text{iff. } T = int \text{ } or \text{ } T \text{ } is \text{ } a \text{ } reference \text{ } type \\ 8 & \text{iff. } T = long \end{cases}$$

Although $space^{arr}$ is a rigid function, terms having $space^{arr}$ as top level function symbol can be non-rigid since the second argument of a $space^{arr}$ term can be non-rigid.

- $T^{DL}_{JML}($`\space(T)`$) := T.$`<size>`,
  where $T.$`<size>` is a static implicit field defined for each type T and denoting the maximum size in bytes of an object of type T (note, that this means that the object's exact type can also be a subtype of T). In original JML `\space` can only be applied to expressions. Applying it to types is a KeY specific extension.

- $T^{DL}_{JML}($`\space(o)`$) := maxSpace(o)$,
  where o is a JML expression of non-primitive type which is not a constructor call. The rigid function $maxSpace$ takes as arguments terms of non-primitive type. Intuitively $maxSpace(o)$ represents the size of o. As long as the exact type of o is unknown, this size can't be determined exactly. We know, however, that due to the meaning of $T.$`<size>` described above,

  $$o \; instanceof \; T \rightarrow maxSpace(o) < T.\texttt{<size>}$$

  holds.

  An axiomatization of $\bullet.$`<size>` and $maxSpace(\bullet)$ is given in Section 13.

- Variable loop level *variable working space* clauses contain a bound variable which is translated to a logic variable. Let `working_space(i) w;` be a variable loop level *working space* clause then:

  $$T^{DL}_{JML}(\texttt{working\_space(i) w}) := T^{DL}_{JML}(w)^{[i/x]}$$

  where x is a integer type logic variable not occurring in $T^{DL}_{JML}(w)$.

11.2.7 KeYJML Semantics for RTSJ Programs

For RTSJ programs it is more relevant to know the memory consumption in specific scopes than the overall heap memory consumption in all scopes since memory allocated in a scope entered by the considered method is (as long as the scope is not used elsewhere) already reclaimed again before the method terminates.

Therefore, we adapt the semantics of JML WCMU specifications employed in RTSJ programs in that respect that `working_space` clauses as well as `\working_space` expressions only refer to the memory consumption in the current scope. In the (infrequent) case that a method, for instance, allocates objects in an outer scope the corresponding memory requirements have to be expressed in the pre- and postcondition. This is illustrated by example 11.2.

Example 11.2 (WCMU Specifications of RTSJ Programs) *Memory allocation within scopes local to the specified method is not observable for the caller of the method and needs thus not to be taken into account by the method specification:*

— Java + JML —

```
/*@ working_space \space(new LTMemory()) + \space(new Runnable());
  @*/
public void compute(String s, MyInteger result){
  ScopedMemory s = new LTMemory(10000);
  s.enter(
    new Runnable(){
      public void run(){
        Term t = new Parser().parseTerm(s);
        result.i = evalTerm(t);
      }
    });
}
```

— Java + JML —

The method `add@(MyList)` *allocates memory in a scope that is not necessarily the current scope, which is taken into account in its* **`requires`** *and* **`ensures`** *clause:*

— Java + JML —

```
private Runnable resizeRunnable = new Runnable(){
        public void run(){ resize();}
      };
...

/*@ public normal_behavior
  @  requires
  @     \memoryArea(this).memoryRemaining()>=\working_space(resize()) &&
  @     \inOuterScope(o, this);
  @  working_space \memoryArea(o)==\currentMemoryArea ?
  @     \working_space(resize()) : 0;
  @  ensures \memoryArea(this).consumed==
  @     \old(\memoryArea(this).consumed)+\working_space(resize());
  @*/
public void add(Object o){
  if(tail.next==null){
    MemoryArea.getMemoryArea(this).enter(resizeRunnable);
  }
  tail.element = o;
  tail = tail.next;
}
```

— Java + JML —

As method-level `working_space` clauses, also loop-level `working_space` clauses refer only to the memory consumption within the current scope. Here the same approach as for method specifications could be taken, i.e., encoding memory consumption in scopes different from the current scopes in the loop invariant as Example 11.3 illustrates.

Example 11.3 *Let us assume the loop depicted in the below code allocates at most* c *bytes (where* c *is a JML expression) in a scope* s *in each of its iterations. This could be specified by a loop invariant as follows:*

Java + JML

```
l: ...
/*@ loop_invariant i>=0 && s.consumed<=i*\old(c, l)+\old(s.consumed,l);
  @ decreasing n-i;
  @ ...
  @*/
for(int i=0; i<n; i++){ ... }
```

Java + JML

Alternatively, we can parameterize the working space clause with the scope in which the specified amount of memory is supposed to be allocated. For the loop from Example 11.3 this results in the following specification:

Java + JML

```
/*@ loop_invariant i>=0;
  @ decreasing n-i;
  @ working_space[s] c;
  @ ...
  @*/
for(int i=0; i<n; i++){ ... }
```

Java + JML

This second approach carries several advantages:

- More compact and less redundant[3] specifications.
- The usage of labels and the "labeled" `\old` construct (as depicted in Example 11.3) is not required. This is especially advantageous in the case of loops with a dependent working space clause which would require employing the `\sum` construct when using the first approach. This could lead to more involved specifications since bound variables (as the one bound by `\sum`) are not allowed to occur in the argument expression of `\old`.

In the following we will not explicitly consider parameterized working space clauses as the proof obligation and taclets (presented in Sections 12 and 13) can be canonically adapted to parameterized working space clauses.

[3] Using the first alternative we need to encode information in the loop invariant that is already entailed by the variant. For instance, in Example 11.3 the variant already states that `i` is an upper bound to the number of already performed loop iterations, yet this information must also be included in the loop invariant.

12 Memory Contracts and Proof Obligations

Since its memory consumption is now considered a relevant aspect of a method's behavior, we have to adapt our notion of a method contract (as introduced in Section 2.3.5) accordingly. A method contract is now a sextuple:

$$(Pre, Post, Mod, Dep, Cap, WS)$$

where WS is a JAVA DL term representing the *working space* clause. Again we may omit irrelevant parts of the contract if possible.

In correspondence to the POs shown in Section 2.3.5 we define now a PO, which we call $RespectsWorkingSpace$, that is valid if and only if a working space clause specifies a correct WCMU upper bound for its specification case:

$$\begin{aligned}&RespectsWorkingSpace(ct; Assumed) :=\\&\quad Conj_{Assumed} \wedge Pre_{ws} \wedge Conj_{InvRealtime} \wedge ValidCall_{op}^{SCJ} \rightarrow\\&\quad \{\mathbf{c}_{max} := \mathbf{c} + WS\}\langle Prg_{op}()\rangle\, \mathbf{c} \leq \mathbf{c}_{max}\end{aligned}$$

where

- $\mathbf{c}$ stands for $defaultMemoryArea.consumed@(MemoryArea)$ (the amount of memory consumed in the current memory area),
- $Conj_{Assumed}$ is defined as in Section 2.3.5,
- $Conj_{InvRealtime}$ is defined as in Section 7.2,
- $\mathbf{c}_{max}$ is a virtual program variable storing the pre-state value of $\mathbf{c} + WS$.
- we assume that there is at least as much space remaining in $defaultMemoryArea$ as maximally need if $Prg_{op}()$ complies with its contract. Therefore:

$$Pre_{ws} := Pre \wedge WS \leq defaultMemoryArea.memoryRemaining()$$

This PO can be applied for checking the correctness of *working space* clauses of RTSJ as well as JAVA and JAVA CARD programs. For this purpose the heap of JAVA and JAVA CARD programs can be considered to be represented by the memory area $defaultMemoryArea$.

When employing the memory-consumption-aware rules shown in Section 13, the correctness and completeness of POs defined so far (in Section 7.2 and 9.2.1) can be preserved by using the strengthened precondition Pre_{ws} instead of Pre in these POs.

Remark 12.1 (Alternative Proof Obligation) *One could argue that it is sufficient to prove termination of $Prg_{op}()$ since surpassing of the specified memory usage WS would potentially result in an `OutOfMemoryError`, as Pre_{ws} only entails*

$$WS \leq defaultMemoryArea.memoryRemaining(),$$

to be raised and thus non-termination of $Prg_{op}()$. This consideration leads to the following PO:

$$\begin{array}{l} RespectsWorkingSpace'(ct; Assumed) := \\ \quad Conj_{Assumed} \wedge Pre_{ws} \wedge Conj_{InvRealtime} \wedge ValidCall_{op}^{SCJ} \rightarrow \\ \quad \{\mathbf{c}_{max} := \mathbf{c} + WS\}\langle Prg_{op}()\rangle\, true \end{array}$$

This PO is, however, also valid for pathological programs catching `OutOfMemoryErrors`. This is problematic since it depends on the JVM implementation from which memory scope the memory for an `OutOfMemoryError` is allocated which is not reflected in the calculus (we model `OutOfMemoryErrors` as pre-allocated). Therefore PO $RespectsWorkingSpace$ features the post condition $\mathbf{c} \leq \mathbf{c}_{max}$ instead which makes $RespectsWorkingSpace$ invalid also in case of a program $Prg_{op}()$ surpassing WS and catching the resulting error[1].

Remark 12.2 (JML and KeYJML) *If we apply the original JML semantics here, meaning that the working space clause is evaluated in the post-state we still have to require that there is enough space remaining in $defaultMemoryArea$ as maximally needed if $Prg_{op}()$ complies to its specification, but now the specified WCMU is evaluated in the post-state. Accordingly we could*

- *either approximate the post-state value of WS obtained by the given contract and require that Pre_{ws} is of the form:*

$$Pre \wedge \{*_i^{Mod}\}Post \wedge (\{*_i^{Mod}\}WS) \leq defaultMemoryArea.memoryRemaining() \tag{12.1}$$

which would lead to a PO

$$\begin{array}{l} Conj_{Assumed} \wedge Pre_{ws} \wedge Conj_{InvRealtime} \wedge ValidCall_{op}^{SCJ} \rightarrow \\ \{\mathbf{c}_{old} := \mathbf{c}\}\langle Prg_{op}()\rangle\, \mathbf{c} \leq \mathbf{c}_{old} + S \end{array} \tag{12.2}$$

- *or use the exact value determined by the implementation of op:*

$$\exists x; \left(\begin{array}{l} Conj_{Assumed} \wedge Pre_{ws} \wedge Conj_{InvRealtime} \wedge ValidCall_{op}^{SCJ} \rightarrow \\ \{\mathbf{c}_{old} := \mathbf{c}\}\langle Prg_{op}()\rangle\, \mathbf{c} \leq \mathbf{c}_{old} + S \wedge x \doteq S \end{array} \right) \tag{12.3}$$

where

$$Pre_{ws} := Pre \wedge x \leq defaultMemoryArea.memoryRemaining()$$

[1] We consider the value of $\mathbf{c}$ to be increased by an allocation even if this allocation raises an `OutOfMemoryError`. See rule allocate in Section 13.2.

13 Calculus

We now turn to calculus rules extending the existing JAVA DL calculus provided by KeY. These newly defined rules reflect the semantics of KeYJML and JAVA DL expressions described in Section 11.2 and make JAVA DL suitable for reasoning with memory performance aspects of JAVA programs. We can basically distinguish two different types of rules: Those that axiomatize the semantics of the new symbols like ws^r and those that compute the memory consumption of a program by symbolic execution.

13.1 Axiomatization of ws^r, ws^{nr}, $space^{arr}$, $maxSpace$ and $C.$<size>

We start with Section 13.1.1 defining the platform independent axioms and corresponding rules holding for the symbols ws^r, ws^{nr}, $space^{arr}$, $maxSpace$ and $C.$<size> before showing in Section 13.1.2 which rules are necessary two reflect the characteristics of a specific target platform.

13.1.1 Platform Independent Rules

We know that a method can have no negative memory consumption which is stated by axiom 13.1:

Axiom 13.1 *For each symbol* $ws^r_{m(\bar{a}),\varphi}$ *and* $ws^{nr}_m(\bar{a})$*:* $ws^r_{m(\bar{a}),\varphi} \geq 0$ *and* $ws^{nr}_m(\bar{a}) \geq 0$ *is valid.*

Axiom 13.1 is reflected in the rules $wsGEqZeroR$ and $wsGEqZeroNR$:

$$\textsf{wsGEqZeroR}\ \frac{\Gamma, ws^r_{m(\bar{a}),\varphi} \geq 0 \ \Rightarrow \Delta}{\Gamma \Rightarrow \Delta} \ni ws^r_{m(\bar{a}),\varphi}$$

$$\textsf{wsGEqZeroNR}\ \frac{\Gamma, \{\mathcal{U}\}ws^{nr}_m(\bar{a}) \geq 0 \ \Rightarrow \Delta}{\Gamma \Rightarrow \Delta} \ni \{\mathcal{U}\}ws^{nr}_m(\bar{a})$$

where $\{\mathcal{U}\}$ is an arbitrary update. As opposed to rule wsGEqZeroNR, in rule wsGEqZeroR we do not need to regard the state in which $ws^r_{m(\bar{a}),\varphi}$ occurs in the sequent the rule is applied to since $ws^r_{m(\bar{a}),\varphi}$ is rigid.

In Section 11.2 we introduced the implicit fields $T.$<size> for each type T and the function symbol $maxSpace$ for specifying the memory consumption of objects with unknown runtime type. We already sketched a semantics for them which is now formalized by the axioms 13.2 and 13.3 and the corresponding calculus rules sizeInstance and sizeSubtype.

Axiom 13.2 *For every term* o *with* $o \in Terms_T$ *and* T *being a reference type the following holds:*

$$maxSpace(o) \leq T.\texttt{<size>}$$

The corresponding rule sizeInstance is only applicable to sequents containing a term of the form $maxSpace(o)$:

$$\text{sizeInstance}\ \frac{\Gamma,\ maxSpace(o) \leq T.\texttt{<size>} \Rightarrow \Delta}{\Gamma \Rightarrow \Delta} \ni maxSpace(o)$$

For all types T_1, T_2 with $T_1 \sqsubseteq T_2$ (T_1 is subtype of T_2) each instance of T_1 is an instance of T_2. This entails that the maximum size over all objects of type T_1 is smaller or equal to the maximum size over all objects of type T_2.

Axiom 13.3 *For all types T_1, T_2 with $T_1 \sqsubseteq T_2$ $T_1.\texttt{<size>} \leq T_2.\texttt{<size>}$ is valid.*

This axiom justifies the following rule:

$$\text{sizeStatic}\ \frac{\Gamma,\ T_1.\texttt{<size>} \leq T_2.\texttt{<size>} \Rightarrow \Delta}{\Gamma \Rightarrow \Delta}$$

Concrete values or upper bounds for the fields $T.\texttt{<size>}$ may be determined by the specification, for instance, by a class invariant of the following form:

— JML —

```
//@ invariant \space(T)<limit;
```

— JML —

The rigid term $ws^r_{m(\bar{a}),\varphi}$ denotes the specified WCMU of method invocation $m(\bar{a})$ under the precondition φ. Thus for two working space terms $ws^r_{m(\bar{a}),\varphi_1}$ and $ws^r_{m(\bar{a}),\varphi_2}$ with $\varphi_1 \rightarrow \varphi_2$ the maximum amount of heap space consumed by m under the precondition φ_1 cannot be larger than under the precondition φ_2 since the set of states satisfying φ_1 is a subset of the set of states satisfying φ_2:

$$\{s | s \models \varphi_1\} \subseteq \{s | s \models \varphi_2\}$$

This manifests itself in the following axiom

Axiom 13.4 *Let φ_1 and φ_2 be* Java DL *formulas and m a method then the following holds:*

If $\varphi_1 \rightarrow \varphi_2$ is valid then $ws^r_{m(\bar{a}),\varphi_1} \leq ws^r_{m(\bar{a}),\varphi_2}$ is valid.

This axiom is encoded in the rule wsRigid:

$$\text{wsRigid}\ \frac{\begin{array}{c}\Gamma \Rightarrow \{*_i\}(\varphi_1 \rightarrow \{\mathcal{V}\}\varphi_2),\ \Delta \\ \Gamma,\ ws^r_{m(\bar{a}),\varphi_1} \leq ws^r_{m(\bar{b}),\varphi_2} \Rightarrow \Delta\end{array}}{\Gamma \Rightarrow \Delta} \ni ws^r_{m(\bar{a}),\varphi_1},\ ws^r_{m(\bar{b}),\varphi_2}$$

where

- $\bar{a} := T_1\ a_1, \ldots, T_n\ a_n$ and $\bar{b} := T_1\ b_1, \ldots, T_n\ b_n$
- the update $\{\mathcal{V}\} := \{b_1 := a_1 || \ldots || b_n := a_n\}$ updates the method parameters occurring in φ_2 with those occurring in φ_1 and thus ensures that in both formulas the parameters of m are named identically.
- $\{*_i\}$ is a fresh (i.e., not yet occurring in Γ, Δ) anonymous update

The reason for applying $\{*_i\}$ in the first premise is that we need to show that $\varphi_1 \to \varphi_2$ is valid and thus holding in every state and not just those constrained by the sequent context Γ and Δ. The update $\{*_i\}$ erases, so to speak, the information on the state determined by the sequent context. In the second premise we can then use $ws^r_{m(\bar{a}),\varphi_1} \leq ws^r_{m(\bar{a}),\varphi_2}$ as an assumption (meaning it becomes part of the antecedent).

The relation between a non-rigid working space term and a rigid one can be defined in similar manner as done by rule wsRigid for two rigid working space terms: If the condition φ holds in a certain state then the value of $ws^r_{m(\bar{a}),\varphi}$ is an upper bound for $ws^{nr}_m(\bar{a})$ when evaluated in this state. This circumstance is formalized by axiom 13.5.

Axiom 13.5 *For all Kripke structures $\mathcal{K} := (\mathcal{M}, \mathcal{S}, *, \rho)$, all states $s \in \mathcal{S}$, all updates $\mathcal{U}$ and all formulas φ the following holds:*

If $t \models \varphi$ with $t := val_s(\mathcal{U})(s)$ then $val_t(ws^{nr}_m(\bar{a}))$ is smaller or equal to $val_s(ws^r_{m(\bar{a}),\varphi})$

We make use of axiom 13.5 in the following rule:

$$\textsf{wsNonRigid}\ \frac{\begin{array}{c}\Gamma \Rightarrow \{\mathcal{U}\}\{\mathcal{V}\}\varphi,\ \Delta \\ \Gamma,\ \{\mathcal{U}\}ws^{nr}_m(\bar{a}) \leq ws^r_{m(\bar{b}),\varphi} \Rightarrow \Delta\end{array}}{\Gamma \Rightarrow \Delta}\ \ni \{\mathcal{U}\}ws^{nr}_m(\bar{a}), ws^r_{m(\bar{b}),\varphi}$$

In the first premise we have to show that φ holds in the state determined by $\mathcal{U}$ and Γ, Δ if the parameters $\bar{b}$ have the values $\bar{a}$ while in the second premise we can then assume that the memory consumption of m in this state is smaller or equal than $ws^r_{m(\bar{a}),\varphi}$ (indicated by $\{\mathcal{U}\}ws^{nr}_m(\bar{a}) \leq ws^r_{m(\bar{a}),\varphi}$ occurring in the antecedent). Again we use an update $\{\mathcal{V}\} := \{b_1 := a_1 || \ldots || b_n := a_n\}$ to initialise the parameters $\bar{b}$ of m with the arguments $\bar{a}$ determined by $ws^{nr}_m(\bar{a})$.

With the rules defined so far, it is not yet possible to put a working space term $ws^r_{m(\bar{a}),\varphi}$ in relation to the working spaces specified by any of m's contracts. However, we know that if there is a contract C for m whose precondition is logically weaker than φ, the semantics of $ws^r_{m(\bar{a}),\varphi}$ entails that $ws^r_{m(\bar{a}),\varphi}$ denotes the WCMU WS specified by C as evaluated in some state satisfying φ.

Axiom 13.6 *If $\varphi \to Pre$ is valid then $s \models ws^r_{m(\bar{a}),\varphi} \doteq WS$ for some state s with $s \models \varphi$ and $val_s(\bar{a}) = val_s(\bar{p})$ with $\bar{p}$ being the placeholders for m's parameters used in C.*

Analogously, axiom 13.7 states that in the case that φ is weaker than C's precondition every value WS, when evaluated in a state satisfying Pre, is a lower bound for $ws^r_{m(\bar{a}),\varphi}$.

Axiom 13.7 *If $Pre \to \varphi$ is valid then $s \models WS \leq ws^r_{m(\bar{a}),\varphi}$ for every state s with $s \models Pre$.*

This results in two calculus rules for the two mentioned cases:

$$\textsf{wsContract1}\ \frac{\begin{array}{c}\Gamma \Rightarrow \{*_i\}(\varphi \to \{\mathcal{V}\}Pre),\ \Delta \\ \Gamma,\ \{*_j\}(\varphi \to ws^r_{m(\bar{a}),\varphi} \doteq \{\mathcal{V}\}WS) \Rightarrow \Delta\end{array}}{\Gamma \Rightarrow \Delta}\ \ni ws^r_{m(\bar{a}),\varphi}$$

$$\textsf{wsContract2}\ \frac{\begin{array}{c}\Gamma \Rightarrow \{*_i\}(\{\mathcal{V}\}Pre \to \varphi),\ \Delta \\ \Gamma,\ \{*_j\}(Pre \to WS \leq ws^r_{m(\bar{a}),\varphi}) \Rightarrow \Delta\end{array}}{\Gamma \Rightarrow \Delta}\ \ni ws^r_{m(\bar{a}),\varphi}$$

Where we use the update $\{\mathcal{V}\} := \{p_1 := a_1 || \ldots || p_n := a_n\}$ sets the parameters $p_1, \ldots, p_n$ used in the regarded method contract to the values of $a_1, \ldots, a_n$. Again, $\{*_i\}$ and $\{*_j\}$ are fresh anonymous updates. The motivation for the anonymous updates $\{*_i\}$ used in each of the above rule's first premises is the same as for rule rigidWS namely that, for instance, the implication $Pre \to \varphi$ (as occurring in rule wsContract2) has to be valid, meaning it is required to hold in every state not only the ones determined by the context formulas Γ and Δ. Since the working space t is only defined for states meeting Pre, we can only assume in the second premise of rule wsContract2 that $WS \leq ws^r_{m,\varphi}$ holds in states also satisfying Pre. This leads us to the formula $\{*\}(Pre \to WS \leq ws^r_{m,\varphi})$ which is part of the antecedent of the second premise of rule wsContract2. The second premise of wsContract1 is motivated analogously except that we can constrain the set of considered states even further by φ (due to $\varphi \to \{\mathcal{V}\}Pre$ being valid and $ws^r_{m,\varphi}$ denoting the worst case memory consumption in states satisfying φ).

For non-rigid working space terms there is a similar axiom motivated basically by the same considerations as axiom 13.6 with WS and Pre as defined above:

Axiom 13.8 *Let $(Pre, \ldots, WS)$ be a contract for m with $\bar{p}$ being the placeholders for m's parameters (and receiver object if necessary). For all Kripke structures $\mathcal{K} := (\mathcal{M}, \mathcal{S}, *, \rho)$, all states $s \in \mathcal{S}$ and all formulas φ the following holds: If $s \models Pre$ and $val_s(\bar{a}) = val_s(\bar{p})$ then $val_s(ws^{nr}_m(a_1, ..., a_n)) = val_s(WS)$.*

The corresponding rule is given by wsContract3:

$$\text{wsContract3}\ \frac{\begin{array}{c}\Gamma \Rightarrow \{\mathcal{U}\}\{\mathcal{V}\}Pre,\ \Delta \\ \Gamma,\ \{\mathcal{U}\}(ws^{nr}_m(\bar{a}) \doteq \{\mathcal{V}\}WS) \Rightarrow \Delta\end{array}}{\Gamma \Rightarrow \Delta} \ni ws^{nr}_m(\bar{a})$$

We use the update $\{\mathcal{V}\} := \{p_1 := a_1 || \ldots || p_n := a_n\}$ to map m's parameters $p_1, \ldots, p_n$ occurring in WS and Pre to the concrete arguments $\bar{a} := a_1, ..., a_n$ taken from $ws^{nr}_m(\bar{a})$. The update $\{\mathcal{U}\}$ represents the state in which the term $ws^{nr}_m(\bar{a})$ occurs in the sequent wsContract3 is applied to.

Remark 13.9 (Soundness of Contract Rules) *For the rules* wsContract1 *and* wsContract3 *to be sound we have to require that for any pair C_1, C_2 of specification cases for a method* m *the condition $\phi_1 \wedge \phi_2 \to w_1 = w_2$ holds, where ϕ_i and w_i denote the precondition and working space of contract C_i. For this condition to be true it is sufficient to require that different specification cases for the same method have disjoint preconditions.*

In case $\phi_1 \wedge \phi_2 \to w_1 \doteq w_2$ does not hold, as, for instance, if we set $\phi_1 := \phi_2 := true$ and $w_1 := 0$, $w_2 := 1$, using wsContract3 *we could prove that the unsatisfiable (according to the semantics of ws^{nr}) formula $ws^{nr}_m() < 0$ holds as these derivation steps illustrate:*

$$\frac{\dfrac{*}{\Rightarrow true, ws^{nr}_m() < 0} \qquad ws^{nr}_m() \doteq 0 \Rightarrow ws^r_{m()} < 0}{\Rightarrow ws^{nr}_m() < 0}$$

By applying wsContract3 *again to the remaining goal $ws^{nr}_m() = 0 \Rightarrow ws^r_{m()} < 0$ using the second contract for m we get a proof tree with one open goal of the form*

$$ws^{nr}_m() \doteq 0, ws^{nr}_m() \doteq 1 \Rightarrow ws^r_{m()} < 0$$

which can eventually also be closed:

$$\cfrac{\cfrac{\cfrac{*}{ws^{nr}_m() \doteq 0, ws^{nr}_m() \doteq 1, false \Rightarrow ws^r_{m()} < 0}}{ws^{nr}_m() \doteq 0, ws^{nr}_m() \doteq 1, 0 \doteq 1 \Rightarrow ws^r_{m()} < 0}}{ws^{nr}_m() \doteq 0, ws^{nr}_m() \doteq 1 \Rightarrow ws^r_{m()} < 0}$$

Remark 13.10 (JML and KeYJML) *Applying the original JML semantics for working space clauses (according to which working space clauses are evaluated in the post-state) leads to the following working space contract rules:*

$$\mathsf{wsContract1'}\ \frac{\begin{array}{l}\Gamma \Rightarrow \{*_i\}(\varphi \to \{\mathcal{V}\}Pre),\ \Delta\\ \Gamma,\ \{*_i\}(\varphi \land \{*_j^{Mod}\}(Post \to ws^r_{m(\bar{a}),\varphi} \doteq \{\mathcal{V}\}WS)) \Rightarrow \Delta\end{array}}{\Gamma \Rightarrow \Delta}\ \ni ws^r_{m(\bar{a}),\varphi}$$

$$\mathsf{wsContract2'}\ \frac{\begin{array}{l}\Gamma \Rightarrow \{*_i\}(\{\mathcal{V}\}Pre \to \varphi),\ \Delta\\ \Gamma,\ \{*_i\}(Post \to WS \leq ws^r_{m(\bar{a}),\varphi}) \Rightarrow \Delta\end{array}}{\Gamma \Rightarrow \Delta}\ \ni ws^r_{m(\bar{a}),\varphi}$$

$$\mathsf{wsContract3'}\ \frac{\begin{array}{l}\Gamma \Rightarrow \{\mathcal{U}\}\{\mathcal{V}\}Pre,\ \Delta\\ \Gamma,\ \{\mathcal{U}\}(\{\mathcal{V}'\}Post \to ws^{nr}_m(\bar{a}) \doteq \{\mathcal{V}'\}WS) \Rightarrow \Delta\end{array}}{\Gamma \Rightarrow \Delta}\ \ni ws^{nr}_m(\bar{a})$$

Where Pre and Post are the pre and postcondition of the applied method contract, t its working space and Mod its assignable clause. In addition we define

$$\mathcal{V}' := \{\mathcal{V}; *_i^{Mod}\}$$

with $\mathcal{V}$ being defined as above.

13.1.2 Platform Specific Rules

We now consider rules constraining the size of objects and arrays with regard to a certain platform (the JVM characteristics exemplarily assumed here are described in Section 11.2.5) or computing the concrete size of an object (array) in case sufficient information is available for this.

The space (measured in bytes) occupied by an array is a multiple of 8 and the overhead (the space occupied by an array without being available to store array elements) of an array is 12 bytes.

Axiom 13.11 *For every $x, y \in \mathbb{N}$ with $x \geq 1$ and $y \geq 0$:*

$$space^{arr}(x, y) = min\{a | a \geq 12 + x * y \land a \equiv 0\ mod\ 8\}$$

Axiom 13.11 is exploited by two rules: The first one (arraySpaceConcreteDim) computes the exact value of a term $space^{arr}(e, l)$ in case the arguments e and l are literals, the second one (arraySizeLowerUpperBound) determines lower and upper bounds for $space^{arr}(e, l)$ and is applicable for arbitrary arguments.

If the length of an array is known (meaning it is a concrete value not only a symbolic expression) the heap space consumed by this array can be determined:

$$\mathsf{arraySpaceConcreteDim}\ \frac{\begin{array}{c}\Gamma \Rightarrow e > 0 \land l \geq 0,\ \Delta \\ [\mathit{space}^{arr}(e,l) \rightsquigarrow \mathit{as}_{e,l}]\end{array}}{\Gamma \Rightarrow \Delta}$$

where

- e and l are integer literals.
- $\mathit{as}_{e,l} := \mathit{min}\{a | a \geq 12 + e * l \land a \equiv 0\ \mathit{mod}\ 8\}$

In case e and l are no literals but only symbolic values, we can still determine upper and lower bounds of the $\mathit{space}^{arr}(e,l)$ term depending on the value of e and l:

$$\mathsf{arraySizeLowerUpperBound}\ \frac{\begin{array}{l}\Gamma \Rightarrow e > 0 \land l \geq 0,\ \Delta \\ \qquad \mathit{space}^{arr}(e,l) \leq \mathit{ub}(e,l) \land \\ \Gamma,\ \mathit{space}^{arr}(e,l) \geq \mathit{lb}(e,l) \land \quad \Rightarrow \Delta \\ \qquad \mathit{space}^{arr}(e,l) \geq \mathit{min}_{as}\end{array}}{\Gamma \Rightarrow \Delta} \ni \mathit{space}^{arr}(e,l)$$

where

- $\mathit{ub}(e,l)$ denotes an upper bound of $\mathit{space}^{arr}(e,l)$ for arbitrary values of l. For the targeted JVM implementation we can choose for instance:

$$\mathit{ub}(e,l) := \begin{cases} 8l + 16, & \text{if } e = 8 \\ e(l-1) + 20, & \text{if } e \in \{1,2,4\} \end{cases}$$

- $\mathit{lb}(e,l)$ denotes a lower bound of $\mathit{space}^{arr}(e,l)$ for arbitrary values of l:

$$\mathit{lb}(e,l) := \begin{cases} 8l + 16, & \text{if } e = 8 \\ el + 12, & \text{if } e \in \{1,2,4\} \end{cases}$$

- $\mathit{min}_{as} := \mathit{space}^{arr}(e,0)$ is the size of an array of length 0 which is for the JVM we consider 16 bytes.

In case the argument of $\mathit{maxSpace}$ is a repository object term $\mathit{get}_T(t)$ we know its exact type and can thus also compute its exact size:

$$\mathsf{sizeInstanceExactType} \qquad \mathit{maxSpace}(\mathit{get}_T(t)) \rightsquigarrow \mathit{space}_T$$

where space_T is an integer literal matching the amount of heap space consumed by an object of exact type T (see Section 11.2.5).

Objects are not stored at arbitrary places in memory but are aligned with certain addresses. We consider an alignment with addresses that are a multiple of 8. For arrays this is already expressed by axiom 13.11. Axiom 13.12 reflects this circumstance for arbitrary objects.

Axiom 13.12 *For all terms t_1, t_2 of type integer, all terms o of reference type and all types T the following holds:*

$$\begin{aligned} maxSpace(o) &\equiv 0\, mod\, 8 \\ T.\texttt{<size>} &\equiv 0\, mod\, 8 \\ space^{arr}(t_1, t_2) &\equiv 0\, mod\, 8 \end{aligned}$$

This is stated by the rule objectAlignment:

$$\text{objectAllignment}\ \frac{\Gamma, mod(oSize,\ 8) \doteq 0 \Rightarrow \Delta}{\Gamma \Rightarrow \Delta} \ni oSize$$

Where $oSize$ is a term of the form $maxSpace(o)$, $T.\texttt{<size>}$ or $space^{arr}(t_1, t_2)$ and mod is an interpreted function symbol representing the modulo operation. If $maxSpace(o)$ or $space^{arr}(t_1, t_2)$ are non-rigid they may not occur behind an update or a modality for the rule to be applicable.

13.2 Symbolic Execution

13.2.1 Object Creation

The symbolic execution of the implicit method `<allocate>` increases **c** by the size of the created object. We adapt the rule allocate already shown in Section 6.2.2 to reflect this circumstance:

$$\text{allocate}\ \frac{\begin{array}{l}\Gamma \Rightarrow \{\mathcal{U}\}\{\mathbf{c} := \mathbf{c} + space_T\}\ \backslash if(\alpha)\backslash then(\{v := get_T(T.\texttt{<ntc>})\ \|\\ \qquad T.\texttt{<ntc>} := T.\texttt{<ntc>} + 1\ \|\\ \qquad get_T(T.\texttt{<ntc>}).\texttt{<c>} := TRUE\ \|\\ \qquad get_T(T.\texttt{<ntc>}).\texttt{<ma>} := m\}\langle \pi\, \omega\rangle\, \phi)\\ \qquad \backslash else(\langle \pi\, \texttt{toome}\, \omega\rangle\, \phi),\ \Delta\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle \pi\ \texttt{v=T.<allocate>();}\ \omega\rangle\, \phi,\ \Delta}$$

Where

- $\mathbf{c} := m.consumed$ where m is the currently active memory area as determined by π.
- $\alpha := m.consumed \leq m.size$
- `toome` := **`throw`** `javax.realtime.RealtimeSystem.OOME;`
- $space_T$ is an integer literal representing the heap space occupied by an object of dynamic type T which is in this case (see Section 11.2.5) $space_T := min\{a | a \geq 8{+}s \wedge a\ mod\ 8 \equiv 0\}$ where s is the space occupied by the fields of the object (for a non-primitive field only the space occupied by the reference, namely 4 bytes, not the object itself). For the case that T is an array type we define $space_T := 0$. The memory consumption caused by the creation of an array is accounted for by rule arrayCreation.

In rule allocate we distinguish the two cases that the available memory in the current memory area m (determined by π, see Section 6.2.1) is

- not sufficient to perform the allocation. In this case an `OutOfMemoryError` is raised.
- is sufficient to perform the allocation. In this no error is raised.

Array constructors are treated in a similar way with the mere difference that the size of an array cannot necessarily be statically determined since it depends on the array's dimension:

arrayCreation

$$\frac{\Gamma \Rightarrow \{\mathcal{U}; \mathbf{c} := \mathbf{c} + T^{DL}_{JML}(\texttt{\textbackslash space(new T[d1]...[dn][]...[])})\}\langle \pi\, \texttt{AC}\, \omega\rangle\, \phi,\ \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}\langle \pi\, \texttt{v=new T[d1]...[dn][]...[]);}\, \omega\rangle\, \phi,\ \Delta}$$

Where `AC` is a placeholder for the code modeling the array's creation (for details on the handling of array creation in Java DL refer to [Beckert et al., 2007]) and

$$T^{DL}_{JML}(\texttt{\textbackslash space(new T[d1]...[dn][]...[])})$$

is defined as in Section 11.2.6. We do not have to consider the case that an `OutOfMemoryError` is thrown this time since, among other things, the program `AC` calls the method `<allocate>` whose symbolic execution will trigger this case distinction.

13.2.2 Method Calls

In order to be also able to modularly verify performance contracts (as it has also been proposed in [Krone et al., 2001]), we need a rule describing the effect (including its memory consumption) of a method's execution merely by utilizing the information retrievable from its specification instead of symbolically executing the method body. In Section 2.3.5 such a rule was already presented, that was, however, not aware of the method's memory consumption. We now extend this rule to reflect also the information obtained from the working space clause. Let $(Pre, Post, Mod, Dep, Cap, WS)$ be a method contract:

methodContractInstMem

$$\frac{\begin{array}{l}\Gamma \Rightarrow \{\mathcal{U}\}\{self := se_{rec} || p_1 := a_1 || \ldots || p_n := a_n\}Pre'_{ws},\ \Delta \\ \Gamma,\ \{*^{Mod}_i\}exc \doteq \texttt{null} \Rightarrow \\ \quad \{\mathcal{U}\}\{self := se_{rec} || p_1 := a_1 || \ldots || p_n := a_n\}(ws^{nr}_m(self, p_1, ..., p_n) \doteq WS \rightarrow \\ \qquad \{*^{Mod}_i || \mathbf{c} := \mathbf{c} + ws^{nr}_m(self, p_1, ..., p_n)\}(Post' \rightarrow \{lhs := result\}\langle \pi \quad \omega\rangle\, \phi)),\ \Delta \\ \Gamma,\ \{*^{Mod}_i\}exc \not\doteq \texttt{null} \Rightarrow \\ \quad \{\mathcal{U}\}\{self := se_{rec} || p_1 := a_1 || \ldots || p_n := a_n\}(ws^{nr}_m(self, p_1, ..., p_n) \doteq WS \rightarrow \\ \qquad \{*^{Mod}_i || \mathbf{c} := \mathbf{c} + ws^{nr}_m(self, p_1, ..., p_n)\}(Post' \rightarrow \langle \pi \ \ throw\ exc;\ \omega\rangle\, \phi)),\ \Delta\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle \pi \ \ lhs = se_{rec}.m(a_1, ..., a_n)@(C);\ \omega\rangle\, \phi,\ \Delta}$$

Where (see also Section 2.3.5):

$$\begin{array}{rcl} Pre'_{ws} & := & Pre_{ws} \wedge Conj_{Assumed} \\ Post' & := & Post \wedge Conj_{Ensured} \end{array}$$

As we can see, methodContractInstMem also describes that in every state s reachable by state update $\mathcal{U}$ the worst case memory consumption of $m()$ (when executed in state s) equals WS evaluated in state s (indicated by the update $\mathbf{c} := \mathbf{c} + ws^{nr}_m(self, p_1, ..., p_n)$ and the term $ws^{nr}_m(self, p_1, ..., p_n) \doteq WS$). A rule for static method calls can be defined analogously.

Remark 13.13 (Working space terms in rule methodContractInstMem**)** *The way working space term $ws^{nr}_{m}(self, p_1, ..., p_n)$ is used in rule* methodContractInstMem *could seem to be counter intuitive at first glance since the rule*

$$
\begin{array}{l}
\mathsf{methodContractInstMem'} \\
\quad \Gamma \Rightarrow \{\mathcal{U}\}\{self := se_{rec} || p_1 := a_1 || \ldots || p_n := a_n\} Pre'_{ws},\ \Delta \\
\quad \Gamma,\ \{*^{Mod}_{i}\} exc \doteq \mathtt{\textbf{null}} \Rightarrow \\
\qquad \{\mathcal{U}\}\{self := se_{rec} || p_1 := a_1 || \ldots || p_n := a_n\}\{*^{Mod}_{i} || \mathbf{c} := \mathbf{c} + WS\} \\
\qquad\quad (Post' \rightarrow \{lhs := result\}\langle \pi \quad \omega\rangle\, \phi),\ \Delta \\
\quad \Gamma,\ \{*^{Mod}_{i}\} exc \not\doteq \mathtt{\textbf{null}} \Rightarrow \\
\qquad \{\mathcal{U}\}\{self := se_{rec} || p_1 := a_1 || \ldots || p_n := a_n\}\{*^{Mod}_{i} || \mathbf{c} := \mathbf{c} + WS\} \\
\qquad\quad (Post' \rightarrow \langle \pi\ \ throw\ exc;\ \omega\rangle\, \phi),\ \Delta \\
\hline
\qquad\qquad \Gamma \Rightarrow \{\mathcal{U}\}\langle \pi\ \ lhs = se_{rec}.m(a_1, ..., a_n)@(C);\ \omega\rangle\, \phi,\ \Delta
\end{array}
$$

also expresses that $\mathbf{c}$ *is increased by the value w specified by the applied contract.*

However, keeping the information explicitly in the sequent that w equals the memory consumption of m when called in the state defined by $\mathcal{U}$ and the sequent context Γ and Δ (as expressed by the subformula $ws^{nr}_{m}(self, p_1, ..., p_n) \doteq WS$ in rule applyContract*) can ease proving tasks later on in case rigid working space terms occur in ϕ. This is mainly owed to the rule* wsNonRigid *allowing to directly relate rigid and non-rigid working space terms referring to the same method.*

Remark 13.14 (JML and KeYJML) *Basing the rule set on the original JML semantics that demands evaluation of the working space clause in the post-state would result in the following contract rule:*

$$
\begin{array}{l}
\mathsf{methodContractInstMemPost} \\
\quad \Gamma \Rightarrow \{\mathcal{U}\}\{self := se_{rec} || p_1 := a_1 || \ldots || p_n := a_n\} Pre''_{ws},\ \Delta \\
\quad \Gamma,\ \{*^{Mod}_{i}\} exc \doteq \mathtt{\textbf{null}} \Rightarrow \\
\qquad \{\mathcal{U}\}\{self := se_{rec} || p_1 := a_1 || \ldots || p_n := a_n || *^{Mod}_{i}\}(ws^{nr}_{m}(self, p_1, ..., p_n) \doteq WS \rightarrow \\
\qquad\quad \{\mathbf{c} := \mathbf{c} + ws^{nr}_{m}(self, p_1, ..., p_n)\}(Post' \rightarrow \{lhs := result\}\langle \pi \quad \omega\rangle\, \phi)),\ \Delta \\
\quad \Gamma,\ \{*^{Mod}_{i}\} exc \not\doteq \mathtt{\textbf{null}} \Rightarrow \\
\qquad \{\mathcal{U}\}\{self := se_{rec} || p_1 := a_1 || \ldots || p_n := a_n || *^{Mod}_{i}\}(ws^{nr}_{m}(self, p_1, ..., p_n) \doteq WS \rightarrow \\
\qquad\quad \{\mathbf{c} := \mathbf{c} + ws^{nr}_{m}(self, p_1, ..., p_n)\}(Post' \rightarrow \langle \pi\ \ throw\ exc;\ \omega\rangle\, \phi)),\ \Delta \\
\hline
\qquad\qquad \Gamma \Rightarrow \{\mathcal{U}\}\langle \pi\ \ lhs = se_{rec}.m(a_1, ..., a_n)@C;\ \omega\rangle\, \phi,\ \Delta
\end{array}
$$

Where

$$Pre''_{ws} := Pre' \wedge \{*^{Mod}_{i}\}Post' \wedge (\{*^{Mod}_{i}\}WS) \leq defaultMemoryArea.memoryRemaining()$$

with Pre' and $Post'$ being defined as in Section 2.3.5.

13.2.3 Loops

As the last step of adapting the Java DL calculus to performance verification needs we will have a look at a loop invariant rule making use of loop annotations as provided by a JML specification. We first consider the special case of a constant *working space* clause and then the more general case of a variable *working space* clause as defined in Section 11.2.4.

For a constant **wssi** clause we can approximate the loop's WCMU at the beginning of an arbitrary iteration by the product of the WCMU of a single iteration (specified by the **wssi** clause) and the number of iterations performed so far:

$$
\begin{array}{l}
\mathsf{loopInvTotalConstWS} \\
\dfrac{\begin{array}{l}
\Gamma \Rightarrow \{\mathcal{U}\}(Inv \land var \geq 0),\ \Delta \\
\Gamma,\ \{\mathcal{U}; c_1 := \mathbf{c}; w_1 := WSL; v_1 := var; *_i^{Mod}; \mathbf{c} := c_1 + w_1 * (v_1 - var)\} \\
\quad (Inv \land ([\pi\ \mathbf{boolean}\ \mathtt{g{=}e;}]g \doteq \mathit{TRUE}) \land var \geq 0) \Rightarrow \\
\quad \{\mathcal{U}; c_1 := \mathbf{c}; w_1 := WSL; v_1 := var; *_i^{Mod}; \\
\qquad \mathbf{c} := c_1 + w_1 * (v_1 - var); c_2 := \mathbf{c}; v_2 := var\}\langle \mathtt{tc(p,e)}\ \rangle\, \psi,\ \Delta \\
\Gamma,\ \{\mathcal{U}\}\{*_j^{Mod} || \mathbf{c} := \mathbf{c} + var * WSL\}Inv \Rightarrow \\
\quad \{\mathcal{U}\}\{*_j^{Mod} || \mathbf{c} := \mathbf{c} + var * WSL\}[\pi\ \mathtt{g{=}e;}](g \doteq \mathit{FALSE} \rightarrow \langle \pi\, \omega\rangle\, \phi),\ \Delta
\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle \pi\, \mathbf{while}\mathtt{(e)\{p\}}\ \omega\rangle\, \phi,\ \Delta}
\end{array}
$$

Where

- Inv is a loop invariant holding at the beginning and the end of each normally terminating (meaning not termination by **break** or an *exception*) loop iteration.
- Mod is an **assignable** clause defining a set of locations modifiable by the loop. Allowing the specification of assignable clauses for loops is a KeY-specific extension of JML.
- var is the loop variant specified by a **decreasing** clause. The term var strictly decreases in each iteration of the loop while remaining greater or equal to 0 thus inducing termination of the loop.
- The term WSL is a WCMU estimation for a single normally terminating iteration of the loop obtained from the, again KeY-specific, **working_space** clause.

The first premise of loopInvTotalConstWS states that the invariant Inv is valid just before the loop is executed the first time.

The second premise states that, in case the loop body terminates normally, the loop invariant is preserved by the loop body and the loop guard[1] and the memory consumption of this loop iteration does not exceed WSL. Here

- $*_i^{Mod}$ is an anonymising update approximating the state change caused by the loop.
- The update $\mathbf{c} := c_1 + w_1 * (v_1 - var)$ increases $\mathbf{c}$ (whose value before the first iteration of the loop is stored in c_1) by the maximal cumulated amount of heap space the loop has potentially consumed in all (normally terminating) iterations executed before the observed iteration. Since v_1 contains the old value of the loop variant before its first iteration, var its current value and w_1 the value of WSL as evaluated before the first loop iteration $w_1 * (v_1 - var)$ is the maximum amount of memory consumed so far by the loop.
- `g` is a fresh program variable used to store the value of `e`.

[1] The loop guard can potentially have side effects.

- `tc(p,e)` is derived from the original loop body and guard that were transformed in a way that, for instance, allows capturing the execution of **`break`** and **`continue`** statements or the raising of an uncaught (i.e., uncaught *within* the loop body) exception. Such events are memorized by fresh program variables exc, b_{break} and b_{cont}.

- $\psi := \psi_{exc} \land \psi_{break} \land \psi_{normal}$
  ψ describes the conditions required to hold after the loop body terminated normally (given by formula ψ_{normal}) or abruptly by an exception (ψ_{exc}) or break (ψ_{break}).

  - $\psi_{normal} := \begin{array}{l}(exc \doteq \mathtt{\textbf{null}} \land b_{break} \doteq \mathit{FALSE} \lor b_{cont} \doteq \mathit{TRUE}) \rightarrow \\ (Inv \land \mathbf{c} - c_2 \leq w_1 \land var < v_2)\end{array}$

    The rationale behind this formula is that if no exception has been thrown and the loop body terminates normally (indicated by the values of e and b_{break}) or a continue statement has been executed (indicated by $b_{cont} \doteq \mathit{TRUE}$) then

    * the invariant holds
    * **c** was increased at most by the pre-state value of WSL and
    * *var* strictly decreased compared to the pre-state of the considered loop iteration.

  - $\psi_{exc} := exc \not\doteq \mathtt{\textbf{null}} \rightarrow \langle \pi\, \mathtt{\textbf{throw}\ exc;}\ \omega \rangle\, \phi$

    If an exception has been thrown the postcondition ϕ must hold after the execution of the rest of the program π **`throw`** `exc;`ω. We inject the statement **`throw`** `exc;` into the remaining code since the exception was not caught in the original implementation of the loop but only by a **`catch`** block added by the transformation `tc` for memorizing uncaught exceptions.

  - $\psi_{break} := b_{break} \doteq \mathit{TRUE} \rightarrow \langle \pi\, \omega \rangle\, \phi$

    In case a **`break`** statement terminates the loop we are in a similar situation as in the exceptional case to that effect that ϕ must be shown to hold after the remaining code is executed.

  Recapitulating one can say that these explanations about ψ also clarify why it is sufficient to specify only the memory consumption of normally terminating loop iterations: All information needed about abruptly terminating iterations is obtained by symbolic execution of the loop body and thus **c** has already been accordingly increased in the state ψ_{exc} and ψ_{break} are evaluated in.

Finally, in the third premise of rule loopInvTotalConstWS we can use the information that the loop invariant holds after the last iteration of the loop (at the end of which $e \doteq \mathit{FALSE}$ holds) and that **c** was increased at this point by the pre-state value of $var * WSL$, which is encoded in an update.

The invariant rule loopInvTotalVarWS making use of the variable working space clause can be defined analogously to loopInvTotalConstWS. Since the memory consumption of the loop as specified by the **`vwssi`** is not constant over different iterations, we cannot simply express it as the product of the term WSL specified by **`vwssi`** and the number of iterations performed (as done in rule loopInvTotalConstWS). Instead the WCMU of the loop at the beginning of an arbitrary iteration is given by the sum of the values of WSL over all loop iteration performed so far (as described in Section 11.2.4). This is done by means of the sum operator introduced in Section 2.3.8.

loopInvTotalVarWS

$$\frac{\begin{array}{l}
\Gamma \Rightarrow \{\mathcal{U}\}(Inv \land var \geq 0),\ \Delta \\
\Gamma,\ \{\mathcal{U}; c_1 := \mathbf{c}; w_1 := WSL; v_1 := var; *_i^{Mod}; \mathbf{c} := c_1 + \mathrm{sum}(x;\ [0, v_1 - var);\ WSL)\} \\
\quad (Inv \land ([\pi\ \mathtt{g=e;}]g \doteq \mathit{TRUE}) \land var \geq 0) \Rightarrow \\
\quad \{\mathcal{U}; c_1 := \mathbf{c}; v_1 := var; *_i^{Mod}; v_2 := var; \\
\qquad \mathbf{c} := c_1 + \mathrm{sum}(x;\ [0, v_1 - v_2);\ WSL); c_2 := \mathbf{c}\}\langle \mathtt{tc(p,e)}\ \rangle\, \psi',\ \Delta \\
\Gamma,\ \{\mathcal{U}\}\{*_j^{Mod} || \mathbf{c} := \mathbf{c} + \mathrm{sum}(x;\ [0, var);\ WSL)\}Inv \Rightarrow \\
\quad \{\mathcal{U}\}\{*_j^{Mod} || \mathbf{c} := \mathbf{c} + \mathrm{sum}(x;\ [0, var);\ WSL)\}[\pi\ \mathtt{g=e;}](g \doteq \mathit{FALSE} \rightarrow \langle \pi\, \omega\rangle\, \phi),\ \Delta \\
\Gamma \Rightarrow \{\mathcal{U}\}(\forall x;\ WSL \doteq \{*_j^{Mod}\}WSL),\ \Delta
\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle \pi\, \mathtt{\textbf{while}(e)\{p\}}\ \omega\rangle\, \phi,\ \Delta}$$

Where x is a logic variable stemming from the translation of the parameter of the **`vwssi`** clause. This rule differs from loopInvTotalConstWS in the following respects:

- Second premise:
  - In the second premise the accumulated WCMU of the preceding loop iterations used to initialise **c** before the considered loop iteration appropriately is given by $\mathrm{sum}(x;\ [0, v_1 - var);\ WSL)$.
  - $\psi' := \psi_{exc} \land \psi_{break} \land \psi'_{normal}$ with ψ_{exc} and ψ_{break} being defined as above and

    $$\psi'_{normal} := \begin{array}{l} (exc \doteq \mathtt{\textbf{null}} \land b_{break} \doteq \mathit{FALSE} \lor b_{cont} \doteq \mathit{TRUE}) \rightarrow \\ (Inv \land \mathbf{c} - c_2 \leq \mathrm{sum}(x;\ [v_1 - v_2, v_1 - var);\ WSL) \land var < v_2) \end{array}$$

    After normal termination of the loop body the accumulated memory consumption of all preceding iterations must not exceed $\mathrm{sum}(x;\ [0, v_1 - var);\ WSL)$ (where the term var is the loop variant and the program variable v_1 memorizes the variants value just before the first iteration of the loop). Therefore, **c** may only have increased by $\mathrm{sum}(x;\ [v_1 - v_2, v_1 - var);\ WSL)$ compared to its value (which is $c_1 + \mathrm{sum}(x;\ [0, v_1 - v_2);\ WSL)$) before the considered iteration. This is expressed by

    $$\mathbf{c} - c_2 \leq \mathrm{sum}(x;\ [v_1 - v_2, v_1 - var);\ WSL) \tag{13.1}$$

    This approach is only sound since WSL (and thus also the term $\mathrm{sum}(x;\ [0, v_1 - v_2);\ WSL)$) does not depend on any location modified by the loop. If we dropped this restriction we could still maintain a sound rule by simply replacing formula (13.1) in the second premise by:

    $$\mathbf{c} \leq c_1 + \mathrm{sum}(x;\ [0, v_1 - var);\ WSL) \tag{13.2}$$

    Proving this would, however, be much more involved and less automatic (compared to proving formula (13.1)) since arbitrary summands of $\mathrm{sum}(x;\ [0, v_1 - var);\ WSL)$ can change compared to their pre-state value.

In case the variant is only decreased by 1 in each iteration, formula 13.1 can be simplified to:

$$\begin{aligned}&\mathbf{c} - c_2 \leq \mathrm{sum}(x;\ [v_1 - v_2, v_1 - var);\ WSL) \rightsquigarrow\\ &\mathbf{c} - c_2 \leq \mathrm{sum}(x;\ [v_1 - v_2, v_1 - (v_2 - 1));\ WSL) \rightsquigarrow\\ &\mathbf{c} - c_2 \leq \mathrm{sum}(x;\ [v_1 - v_2, v_1 - v_2 + 1);\ WSL) \rightsquigarrow\\ &\mathbf{c} - c_2 \leq WSL^{[x/v_1 - v_2]}\end{aligned}$$

- Third premise: After termination of the loop $\mathbf{c}$ is increased by the maximum amount of memory consumed by the loop which is $\mathrm{sum}(x;\ [0, var);\ WSL)$.
- Fourth premise: We need to show that WSL does indeed not depend on any location in Mod which is done by proving $\{\mathcal{U}\}(\forall x;\ WSL \doteq \{*_j^{Mod}\}WSL)$.

Remark 13.15 *As location dependencies and their corresponding proof obligations showed to be a source of subtle pitfalls in Section 9, let us briefly convince ourselves why the formula*

$$\{\mathcal{U}\}(\forall x;\ WSL \doteq \{*_j^{Mod}\}WSL)$$

can be used to show that WSL does not depend on any location in the loop's assignable clause.

For the soundness of loopInvTotalVarWS *it is sufficient that WSL does not depend on any location in Mod in the symbolic state (which is defined by the sequent context Γ and Δ and the update $\mathcal{U}$) the rule* loopInvTotalVarWS *is applied in.*

We now assume the that there is a state t satisfying the constraints defined by the sequent context and the update $\mathcal{U}$ such that WSL depends on at least one location in $val_t(Mod)$ and show that this entails invalidity of the sequent resulting from the fourth premise of rule loopInvTotalVarWS*:*

Let us assume that there is a state s, a variable assignment β with

$$s, \beta \models \bigwedge_{\gamma \in \Gamma} \gamma \wedge \bigwedge_{\delta \in \Delta} \neg\delta \tag{13.3}$$

and a location $(f, (\bar{a})) \in val_{t,\beta}(Mod)$ (where we define $t := val_{s,\beta}(\mathcal{U})(s)$) such that WSL depends on $(f, (\bar{a}))$ (i.e., $(f, (\bar{a})) \in val_{t,\beta}(Dep)$ for each correct depends clause Dep of WSL). This means that there is a state t', a variable assignment β' (with $\beta'(y) = \beta(y)$ f. a. $y \neq x$) and a value v with

$$val_{t,\beta}(WSL) \neq val_{t',\beta}(WSL) \tag{13.4}$$

and

$$val_{t'}(g)(\bar{b}) = \begin{cases} v & \text{if } g = f \text{ and } \bar{b} = \bar{a} \\ val_t(g)(\bar{b}) & \text{otherwise} \end{cases} \tag{13.5}$$

for all location (and virtual location) function symbols g and all tuples of values $\bar{b}$ complying with g's signature[2].

Then there is a Kripke structure $\mathcal{K} := (\mathcal{M}, \mathcal{S}, *, \rho)$ with $s, t, t' \in \mathcal{S}$ and $t' = *(j)$. Together with equations (13.4) and (13.5) this implies that

$$t, \beta \not\models \forall x;\ WSL \doteq \{*_j^{Mod}\}WSL$$

[2] The existence of such a state t' entails that $(f, (\bar{a})) \in val_{t,\beta}(Dep_{WSL})$ where Dep_{WSL} denotes the depends clause of WSL.

and thus (due to equation (13.3) and the definition of t)

$$s, \beta \not\models \bigwedge_{\gamma \in \Gamma} \gamma \rightarrow \{\mathcal{U}\}(\forall x;\ WSL \doteq \{*_j^{Mod}\}WSL) \vee \bigvee_{\delta \in \Delta} \delta$$

which renders the sequent resulting from the fourth premise of rule loopInvTotalVarWS *invalid.*

For the sake of simplicity and readability, both of the presented loop invariant rules (loopInvTotalConstWS and loopInvTotalVarWS) do not take into account parameterized working space clauses as introduced in Section 11.2.7. As a non-parameterized loop-level working space clause can be considered syntactic sugar for a working space clause parameterized with the currently active scope (`<cma>`), the adaptations for handling "other" parameterized working space clauses are canonical and therefore not discussed in detail here.

14 Examples

In the following we demonstrate the capabilities of the presented approach based on two examples. In the first one we examine modular verification of performance constraints. The second one shows the JML specification of a realistic piece of code with performance specifications that can be verified by KeY fully automatically.

14.1 Modular vs. Non-Modular Verification

We will now regard the modular verification of performance contracts using the example implementations and specifications of the methods `freshInstance`, `clear` and `getInstance` already discussed in Section 11.1. Due to the specification bugs identified for `freshInstance`, its specification is corrected using the rigid **`working_space`** construct in the following way:

— Java + JML —

```
/*@  requires _instance!=null;
  @  working_space \working_space(clear()) +
  @           \working_space(getInstance(), SomeClass._instance==null);
  @*/
public static SomeClass freshInstance(){
    clear();
    return getInstance();
}
```

— Java + JML —

The POs derived from the contracts for `clear` and `getInstance` can be proven automatically. The PO for the first of `getInstance`'s specification cases, for instance, is given by the formula:

$$\begin{aligned}&Conj_{InvRealtime} \wedge ValidCall_{op}^{SCJ} \wedge SomeClass.\_instance \doteq \mathtt{\textbf{null}} \wedge 16 \leq memRem \rightarrow\\ &\quad \{\mathbf{c}_{max} := \mathbf{c} + 16\}\langle \mathtt{res=SomeClass.getInstance();} \rangle\, \mathbf{c} \leq \mathbf{c}_{max}\end{aligned}$$

where $memRem := defaultMemoryArea.memoryRemaining()$. For proving the PO

$$\begin{aligned}&Conj_{InvRealtime} \wedge ValidCall_{op}^{SCJ} \wedge SomeClass.\_instance \not\doteq null \wedge\\ &ws^{nr}_{SomeClass.clear()}() + ws^{r}_{SomeClass.getInstance(),SomeClass.\_instance \doteq \mathtt{\textbf{null}}} \leq memRem \rightarrow\\ &\quad \{\mathbf{c}_{max} := \mathbf{c} + ws^{nr}_{SomeClass.clear()}() + ws^{r}_{SomeClass.getInstance(),SomeClass.\_instance \doteq \mathtt{\textbf{null}}}\}\\ &\quad \langle \mathtt{res=SomeClass.freshInstance();} \rangle\, \mathbf{c} \leq \mathbf{c}_{max}\end{aligned}$$

generated for `freshInstance`'s contract we can either choose to replace the method calls `clear()` and `getInstance()` (encountered when symbolically executing `getInstance`'s method body) by their implementation and symbolically execute them in the following (non-modular verification) or to use the already verified performance contracts of the called methods to approximate the methods' behaviour (modular verification).

14.1.1 Non-Modular Verification

When evaluating the method body of `freshInstance` by executing the method bodies of `clear` and `getInstance` sequentially, we get (after the symbolic execution has terminated) a sequent

$$\begin{aligned}\Gamma \;\Rightarrow\; & \mathit{SomeClass.\_instance} \doteq \texttt{null}, \\ & \mathbf{c} + 16 \leq \mathbf{c} + ws^{nr}_{\mathit{SomeClass.clear()}}() + ws^{r}_{\mathit{getInstance(),SomeClass.\_instance} \doteq \texttt{null}}\end{aligned} \quad (14.1)$$

which shows that $\mathbf{c}$ was increased by 16 bytes[1] and we now have to prove that this is smaller than the sum of the maximum specified amounts of space consumed by the method `clear` and the method `getInstance` when called in a state satisfying $\mathit{SomeClass.\_instance} \doteq \texttt{null}$. This can be achieved by applying the rules wsGEqZeroNR to $ws^{nr}_{\mathit{SomeClass.clear()}}()$ and wsContract1 to $ws^{r}_{\mathit{getInstance(),SomeClass.\_instance} \doteq \texttt{null}}$ using the first contract of `getInstance`. The resulting sequents (where Δ denotes the antecedent of sequent (14.1)) are

$$\Rightarrow \quad \{*\}(\mathit{SomeClass.\_instance} \doteq \texttt{null} \rightarrow \mathit{SomeClass.\_instance} \doteq \texttt{null}), \Delta$$

expressing that $\mathit{SomeClass.\_instance} \doteq \texttt{null}$ has to imply the precondition (which is also given by $\mathit{SomeClass.\_instance} \doteq \texttt{null}$) of the applied contract and

$$\begin{aligned} & \{*\}(\mathit{SomeClass.\_instance} \doteq \texttt{null} \wedge ws^{r}_{\mathit{getInstance(),SomeClass.\_instance} \doteq \texttt{null}} \doteq 16), \\ & 0 \leq ws^{nr}_{\mathit{SomeClass.clear()}}() \\ \Rightarrow \quad & \Delta \end{aligned}$$

containing the assumptions

$$\{*\}(\mathit{SomeClass.\_instance} \doteq \texttt{null} \wedge ws^{r}_{\mathit{getInstance(),SomeClass.\_instance} \doteq \texttt{null}} \doteq 16)$$

and

$$0 \leq ws^{nr}_{\mathit{SomeClass.clear()}}()$$

obtained from the application of wsContract1 and wsGEqZeroNR. Both sequents can be proven valid automatically.

14.1.2 Modular Verification

If we decide to verify the PO (14.1) in a modular way by utilizing `clears`'s and `getInstance`'s contracts by applying methodContractInstMem the symbolic execution of `freshInstance` leads to $\mathbf{c}$ being increased by $\{\mathcal{U}_1\}ws^{nr}_{\mathit{clear()}}()$ and $\{\mathcal{U}_2\}ws^{nr}_{\mathit{getInstance()}}()$ (where $\mathcal{U}_1$ and $\mathcal{U}_2$ are updates representing the symbolic state in which the respective methods where invoked). Thus after the symbolic execution of `freshInstance` is completed we have obtained a sequent:

$$\begin{aligned} & \{\mathcal{U}_1\}ws^{nr}_{\mathit{clear()}}() \doteq 0,\ \{\mathcal{U}_2\}ws^{nr}_{\mathit{getInstance()}}() \doteq 16 \\ \Rightarrow & \\ & \mathit{SomeClass.\_instance} \doteq \texttt{null}, \\ & \mathbf{c} + 16 \leq \mathbf{c} + ws^{nr}_{\mathit{SomeClass.clear()}}() + ws^{r}_{\mathit{getInstance(),SomeClass.\_instance} \doteq \texttt{null}} \end{aligned} \quad (14.2)$$

[1] This results from the increase of $\mathbf{c}$ by the symbolic execution of the constructor call in `getInstance`'s method body.

This proof goal could be discharged in the same manner as in the previous case by applying wsContract1 and wsGEqZeroNR. Another option is, however, due to the presence of the assumption $\{\mathcal{U}_2\}ws^{nr}_{getInstance()}() \doteq 16$, to apply the rule wsNonRigid to the terms $\{\mathcal{U}_2\}ws^{nr}_{getInstance()}()$ and $ws^{r}_{getInstance(),SomeClass.\_instance \doteq \mathtt{null}}$ resulting in the sequents

$$\Gamma \Rightarrow \{\mathcal{U}_2\}(SomeClass.\_instance \doteq \mathtt{null}), \Delta \qquad (14.3)$$

stating that the precondition $SomeClass.\_instance \doteq \mathtt{null}$ has to hold in the symbolic state in which the method call of `getInstance` was evaluated and

$$\Gamma,\ \{\mathcal{U}_2\}ws^{nr}_{getInstance()}() \leq ws^{r}_{getInstance(),SomeClass.\_instance \doteq \mathtt{null}} \Rightarrow \Delta \qquad (14.4)$$

with Γ and Δ being the antecedent and succedent of sequent (14.2). The sequent (14.3) is proven valid automatically, (14.4) requires an interactive application of wsGEqZeroNR as seen in Section 14.1.1 before it is provable automatically.

14.2 Javolution

Javolution [Dautelle, 2009] is a real-time JAVA library facilitating the development of real-time compliant JAVA applications. This is accomplished by, for instance, reducing the need for garbage collection by using so-called memory contexts in which objects, that are no longer needed, are recycled and can be reused the next time an object of the corresponding class is needed in the containing context. Javolution also provides time-deterministic implementations of standard JAVA packages such as collection and map data structures. The example we now consider is taken from the Javolution class `FastMap` which implements the same functionality as `java.util.HashMap` but shows a more time-deterministic behavior. Since possessing a deterministic memory performance is essential for real-time applications, this example also illustrates the suitability of the presented approach for the field of application it is intended for.

The memory performance of the method `setup` shown in Figure 14.1 which is part of the Javolution [Dautelle, 2009] library was specified in JML. Since the setup method is used by `FastMap`'s constructors to create and initialise a new map, its memory performance is of particular relevance for determining the memory consumption of instances of this class. The specification cases can be verified fully automatically by the KeY system. This demonstrates the potential of the presented approach even for rather complex code in realistic real-time applications.

Remark 14.1 *Given properties for a concrete JVM one could simplify the performance specification shown in the example. For instance with the JVM parameters we assume (see Section 11.2.5) the working space expressions*

```
\space(new Entry[1][1<<R0])+(2+capacity)*\space(new Entry())
```

can be simplified to a better human readable

```
240+capacity*40
```

This simplified expression does not need to be computed manually but can be retrieved from the KeY proof for the corresponding contract.

Java + JML

```
/*@ public normal_behavior
  @  requires capacity <= (1 << R0) && capacity>=0;
  @  working_space \space(new Entry[1][1<<R0]) +
  @          (2+capacity)*\space(new Entry());
  @ also public normal_behavior
  @  requires capacity > (1 << R0) && capacity < (1<<30);
  @  working_space \space(new Entry[(2*capacity)>> R0][1<<R0])+
  @      (2+capacity)*\space(new Entry());
  @*/
private void setup(int capacity) {
    int tableLength = 1 << R0;
    /*@ loop_invariant 1 << R0 < capacity ?
      @   tableLength>=1 << R0 && tableLength<2*capacity :
      @   tableLength == 1 << R0;
      @ decreases 1 << R0 < capacity ? 2*capacity-2-tableLength : 0;
      @ assignable tableLength;
      @ working_space 0;          @*/
    while (tableLength < capacity){ tableLength <<= 1;}
    int size = tableLength >> R0;
    _entries = (Entry[][]) new Entry[size][];
    /*@ loop_invariant i>=0;
      @ decreases _entries.length-i;
      @ assignable i, _entries[*];
      @ working_space \space(new Entry[1 << R0]);     @*/
    for (int i=0; i < _entries.length;) {
        int blockLength = 1 << R0;
        _entries[i++] = (Entry[]) new Entry[blockLength];
    }
    _head = new Entry();
    _tail = new Entry();
    _head._next = _tail;
    _tail._previous = _head;
    Entry previous = _tail;
    /*@ loop_invariant i>=0 && previous!=null;
      @ decreases capacity-i;
      @ assignable _tail._next, i;
      @ working_space \space(new Entry());     @*/
    for(int i = 0; i++ < capacity;) {
        Entry newEntry = new Entry();
        newEntry._previous = previous;
        previous._next = newEntry;
        previous = newEntry;
    }
}
```

Java + JML

Figure 14.1: A JML-annotated method from the *Javolution* class `FastMap`

15 Verification of Scope Sizes in the PERC Pico Memory Model

In Section 8.1 modifications of the calculus described in Section 6 are presented for treating runtime checks (for illegal assignments) in a MIB MM. We now regard what adaptations are needed in addition for performing verification of memory contracts in a MIB MM in general and the PERC Pico MM in particular. The changes are introduced stepwise: First we consider the simple MIB MM defined in [Kwon and Wellings, 2004] requiring only marginal adaptations and then extend the approach to PERC Pico.

We first need to make the approach described in Section 8.1 aware of scope sizes, which means when scopes are created (i.e., when methods are invoked) the size of a scope has to be set according to its specification, when objects are allocated the size of the object has to be accounted for by increasing the attribute `consumed@(MemoryArea)` of the appropriate scope accordingly and the POs shown in Section 12 have to be adapted to the MIB MM.

The rule scopedMemoryMethodBodyExpand defined in Section 8.1 describes the creation of local scopes in a MIB MM. The rule scopedMemoryMethodBodyExpandMC represents a slight modification of this rule now initialising also the fields `consumed@(MemoryArea)` (with 0) and `size@(MemoryArea)` (according to the specified memory consumption). The changes in comparison to rule scopedMemoryMethodBodyExpand in Section 8.1 are printed bold:

scopedMemoryMethodBodyExpandMC

$$\frac{\begin{array}{l}
\Gamma,\ \{\mathcal{U}\}(cma.\texttt{stack} \preceq get_{MemoryStack}(j) \land j \geq MemoryStack.<\texttt{ntc}>) \Rightarrow \\
\quad \{\mathcal{U}\}\{get_{LTMemory}(LTMemory.<\texttt{ntc}>).\texttt{stack} := get_{LTMemory}(j) \mid\mid \\
\qquad newMem := get_{LTMemory}(LTMemory.<\texttt{ntc}>) \mid\mid \\
\qquad get_{LTMemory}(LTMemory.<\texttt{ntc}>).<\texttt{c}> := TRUE \mid\mid \\
\qquad \mathbf{get}_{\mathbf{LTMemory}}(\mathbf{LTMemory}.<\mathbf{ntc}>).\mathbf{consumed} := \mathbf{0} \mid\mid \\
\qquad \mathbf{get}_{\mathbf{LTMemory}}(\mathbf{LTMemory}.<\mathbf{ntc}>).\mathbf{size} := \mathbf{localScopeSize} \mid\mid \\
\qquad LTMemory.<\texttt{ntc}> := LTMemory.<\texttt{ntc}> + 1 \mid\mid \\
\qquad MemoryStack.<\texttt{ntc}> := j + 1 \mid\mid \\
\qquad for\ i;\ if(MemoryStack.<\texttt{ntc}> \leq i \land i \leq j) \\
\qquad\qquad\qquad\qquad get_{MemoryStack}(i).<\texttt{c}> := TRUE \\
\qquad \}\langle\pi\ \texttt{method-frame(}\ \begin{array}[t]{l}\texttt{result->lhs,} \\ \texttt{source=C,} \\ \mathbf{\texttt{this}}\texttt{=se,} \\ \texttt{<cma>=newMem,} \\ \texttt{<oma>=mem)} : \{\texttt{body}\}\ \omega\rangle\phi,\ \Delta \end{array}
\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle\pi\ \texttt{lhs=se.meth(args)@scope@(C);}\ \omega\rangle\ \phi,\ \Delta}$$

The value *mem* is again determined by the annotation `@scope` (attached to the method invocation) and the innermost method frame in π.

The rule $\mathsf{allocateMIB}$ describing object allocation in a MIB MM increases the `consumed` attribute of the target scope:

$$\mathsf{allocateMIB}\ \frac{\begin{array}{l}\Gamma \Rightarrow \{\mathcal{U}\}\{\mathbf{c} := \mathbf{c} + \mathit{space}_T\} \\ \qquad \backslash if(\alpha)\backslash then(\{v := \mathit{get}_T(T.\texttt{<ntc>}) \ || \\ \qquad\qquad T.\texttt{<ntc>} := T.\texttt{<ntc>} + 1 \ || \\ \qquad\qquad \mathit{get}_T(T.\texttt{<ntc>}).\texttt{<c>} := \mathit{TRUE} \ || \\ \qquad\qquad \mathit{get}_T(T.\texttt{<ntc>}).\,\texttt{<ma>} := \texttt{mem}\}\langle \pi\, \omega\rangle\, \phi) \\ \qquad \backslash else(\langle \pi\, \texttt{toome}\, \omega\rangle\, \phi),\ \Delta\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle \pi\ \texttt{v=T.<allocate>()@scope;}\ \omega\rangle\, \phi,\ \Delta}$$

Where

- $\mathbf{c} := \texttt{mem.consumed}$
- $\alpha := \texttt{mem.consumed} + \mathit{space}_T \leq \texttt{mem.size}$
- *mem* is defined as for rule scopedMemoryMethodBodyExpandMC
- `toome` and space_T are defined as for rule allocate in Section 13

The PO *RespectsWorkingSpace* as defined in Section 12 states that memory consumption in the current scope does not surpass a certain specified upper bound. Since we are now interested in the memory consumption taking place in the local scope of the method we reason about, we define a new PO *RespectsLocalScopeSize* in a similar manner.

$$\begin{array}{l} \mathit{RespectsLocalScopeSize}(ct; \mathit{Assumed}) := \\ \qquad \mathit{Conj}_{\mathit{Assumed}} \wedge \mathit{Pre} \wedge \mathit{Conj}_{\mathit{InvRealtime}} \wedge \mathit{ValidCall}^{SCJ}_{op} \rightarrow \\ \qquad \{\mathit{LTMemory}.\texttt{<ntc>}^{@pre} := \mathit{LTMemory}.\texttt{<ntc>}\}\langle \mathit{Prg}_{op}()\rangle\, \mathit{ls.consumed} \leq \mathit{ls.size} \end{array}$$

where

- ls abbreviates $\mathit{get}_{\mathit{LTMemory}}(\mathit{LTMemory}.\texttt{<ntc>}^{@pre})$ which is the local scope of the method call embedded in $\mathit{Prg}_{op}()$,
- $\mathit{Conj}_{\mathit{Assumed}}$ is defined as in Section 2.3.5,
- Pre is defined as in Section 2.3.5,
- $\mathit{Conj}_{\mathit{InvRealtime}}$ and $\mathit{ValidCall}^{SCJ}_{op}$ are defined as in Section 7.2.

Both, [Kwon and Wellings, 2004] and [Nilsen, 2006], provide means to define the size of local scopes. This can for instance be done by specifying the size of the local scope in bytes or (similar to what can be done with the **`\space`** construct in JML specifications) by providing the types of objects created and the respective number of instances for each type. These specifications can be canonically expressed in Java DL. Figure 15.1 shows a method augmented with PERC Pico annotations for determining the *local* and *constructed*[1] scope sizes. This example also hints why the approach defined so far is not sufficient for verifying memory contracts of PERC Pico programs. So far we considered a *constructed* scope to be identical with

[1] In this case the *constructed* scope is associated with the returned object.

— PERC Pico —

```
@ScopedMemorySize(instances = {1}, types = {Object.class})
@ConstructedScopedMemorySize(instances = {1}, types = {Bar.class})
@CallerAllocatedResult
public Foo getFoo(){
  Object tmp = new Object();
  Bar bar = new Bar();
  Foo foo = new Foo();
  foo.bar = bar;
  return foo;
}
```

— PERC Pico —

Figure 15.1: PERC Pico Annotations

the scope of its associated object which is sound for modelling illegal-assignment-checks. For determining whether scope sizes are sufficient it is necessary to know that *constructed* scopes are physically different from other scopes on the same scope stack level. Section 15.1 gives an overview of PERC Pico annotations as far as needed for understanding Section 15.2 which elaborates on the handling of *constructed* scopes. Section 15.3 provides a proof obligations showing that constructed scopes PERC Pico programs are sufficiently sized. Finally Section 15.4.1 shortly summarizes how the calculus in Section 13 needs to be modified to suit the PERC Pico memory model.

15.1 PERC Pico Memory Annotations

This section provides an overview of the semantics of different kinds of scoped memory areas distinguished by PERC Pico and a selection of the respective annotation means provided for them. For an exhaustive account of PERC Pico's annotations refer to [Aonix, 2008].

15.1.1 Local Scopes

Local (method) scopes are associated with method invocations. Their lifetime matches the duration of the associated method execution. The size of a method's local scope is either determined by a

- static analysis providing a conservative approximation of the required scope size. For applying this analysis to a certain method, this method has to be marked with the

  — PERC Pico —

  ```
  @StaticAnalyzable
  ```

  — PERC Pico —

  annotation. Performing the static analysis (which is not always feasible) requires additional helper annotations (such as loop bounds) to support the analysis.

- user-provided annotations attached to the corresponding method or constructor declaration. These annotations can be of the form

  — PERC Pico —

  ```
  @ScopedMemorySize(bytes=k)
  ```

  — PERC Pico —

  providing the size of the local scope measured in bytes or

  — PERC Pico —

  ```
  @ScopedMemorySize(instances = {h1,...,hn}, types = {T1,...,Tm},
                    primitive_array_instances = {j1,..., jn},
                    primitive_array_lengths = {lp1,..., lpn},
                    primitive_array_types = {PT1,...,PTn},
                    reference_array_instances = {k1,..., ks},
                    reference_array_lengths = {lr1,..., lrs},
                    reference_array_types = {RT1,...,RTs},
                    constructed_scopes = cs)
  )
  ```

  — PERC Pico —

  stating that the local scope size is

  $$\sum_{i=1}^{m} \mathtt{ki} * space_{\mathtt{Ti}} + \sum_{i=1}^{n} \mathtt{ji} * space_{\mathtt{PTi[lpi]}} + \sum_{i=1}^{s} \mathtt{ki} * space_{\mathtt{RTi[lri]}} + cs * space_{consSc}$$

  where $space_{\mathtt{T}}$ denotes the size of an object of exact type `T` and $space_{\mathtt{T[l]}}$ the size of the array **`new`** `T[l]`. Empty clauses (occurring for instance in the case that no arrays are created) can be omitted in the above annotation. Local scopes of constructors and methods annotated with **`@CallerAllocatedResult`** or **`@CallerAllocatedArrayResult`** need to set aside memory for the constructed scopes of objects allocated within the method's local scope. The maximum number of such constructed scopes nested within the method's local scope is determined by the **`constructed_scopes`** clause.

15.1.2 Immortal Memory

Immortal memory has application lifetime. As for RTSJ programs, we can consider *immortal* memory to be the primordial scope (i.e., being an outer scope of each other scope existing within the regarded application). Note, that this does not mean that it is equal to the local scope of the application's `main` method. Methods performing immortal memory allocation must be annotated with the

— PERC Pico —

```
@ImmortalAllocation
```

— PERC Pico —

annotation. There are, however, no annotations for specifying the amount of memory allocated within immortal memory. The size of the immortal memory area is determined by a configuration file entry.

15.1.3 Constructed Scopes

Constructed scopes are associated with an object and share the same lifetime with this object. As for local scopes, the size of a constructed scope is either determined by static analysis or by an annotation

— PERC Pico —

```
@ConstructedScopedMemorySize(...)
```

— PERC Pico —

attached to a method (in this case the constructed scope is associated with the returned object) or constructor (in this case the constructed scope is associated with the object initialised by the constructor) declaration. The size of the constructed scope is specified in the same way as for the **@ScopedMemorySize** annotation. Objects can only be allocated in a constructed scope during execution of the corresponding method/constructor creating the constructed scope. Note, that this entails that for each execution context there is at most one constructed scope (which is different from the currently executed method's receiver object's *reentrant* scope, see Section 15.1.4) available (in which the **new** operator can allocate memory), namely the one created when the currently executing method was invoked.

Methods allocating the returned object in the caller context must also be annotated with the **@CallerAllocatedResult** annotation (see Section 8.1).

Figure 15.1 illustrates the usage of constructed scopes: The object created by the constructor invocation **new** Foo() (line 7) is allocated in the caller context of getFoo which is determined by the **@CallerAllocatedResult** (line 3) annotation. The size of the returned object is not accounted for in the annotation of getFoo(). It is the responsibility of the caller to provide enough memory to allocate it. The field foo.bar (line 8) is initialised with a newly created object of type Bar which therefore needs to be allocated in the constructed scope of foo. The annotation (line 2)

— PERC Pico —

```
@ConstructedScopedMemorySize(instances = {1}, types = {Bar.class})
```

— PERC Pico —

ensures that this *constructed* scope is sufficiently big. The caller of a method does in general (except for constructed scopes of objects allocated in (i) a *reentrant* scope or (ii) the local scope of constructors or **@CallerAllocatedResult** methods) not need to set aside memory for the *constructed* scope of the object returned by the called method. The object allocated by **new** Object() (line 5) is never assigned to a reference outliving the method execution and can thus be allocated in the local scope of getFoo.

A constructed scope is physically different from the scope containing its associated object (even though the lifetimes of these two scopes are the same). By using the

— PERC Pico —

```
@ExternallyConstructedScope
```

— PERC Pico —

it is possible to avoid the overhead for creating such a new scope and use the scope containing the associated object instead. The caller of a method/constructor annotated with this annotation has to provide sufficient memory not only for the returned object itself but also for newly created objects referenced by this object and allocated in its (externally-) constructed scope.

15.1.4 Reentrant Scopes

Reentrant scopes are a variant of *constructed* scopes. If a class declaration is annotated with **@ReentrantScope**, each instance method of this class can allocate objects in the constructed scope of its receiver object. Constructed scopes of objects allocated within a *reentrant* scope are embedded within the *reentrant* scope entailing that the *reentrant* scope has to set aside the space for objects allocated in these constructed scopes and some space required as overhead for the constructed scopes themselves. One typical application for *reentrant* scopes are data structures like lists or maps: Entry objects of these data structures need to possess at least the same lifetime as the data structure itself and are therefore allocated in its *reentrant* scope.

The constructed scope associated with an object is in general not uniquely determined: In case an object is initialised by a chain of constructor calls (as realized by calls to **this** or **super** constructors) not annotated with **@ExternallyConstructedScope** each constructor creates its own constructed scope. If there are multiple constructed scopes created for an instance of a class annotated with **@ReentrantScope** they are merged into a single constructed scope of accumulated size, as stated by the PERC Pico manual [Aonix, 2008]:

> If multiple chained constructor invocations for a **@ReentrantScope** ... class each introduce a constructed scope, the effect of object construction is to create a single constructed scope of the accumulated size represented by adding the individual constructed scope sizes.

15.2 Treatment of Constructed Scopes

For each execution context there are now up to five scopes available in which a newly created object can potentially be allocated:

- The *immortal* memory area which can be obtained by `ImmortalMemory.instance()`
- The current local scope accessible by the pointer `<cma>`
- The (not necessarily direct) caller's scope, given by the pointer `<oma>`
- The constructed scope which is for each execution context uniquely determined (see Section 15.1.3). To be able to explicitly reference also this memory area we introduce a pointer `<coma>` (shorthand for *constructed memory area*).
- The reentrant scope associated with the receiver object of the currently executed method. We introduce an implicit instance field `<rs>@Object` referring to this scope.

As for `<cma>` and `<oma>`, the value of this pointer is determined by the (once more extended) method frame statement

Java

```
method-frame(result->retvar,
             source=T,
          this=self,
          <cma>=cm,
          <oma>=om,
          <coma>=com) : {body}
```

Java

As in Section 8.1, we assume that the results[2] of the static analysis performed by PERC Pico for determining the appropriate allocation context are back-annotated by a program transformation. We refer to this transformation as T_{PERC} in the following. Analogously to the transformation T_{MIB} used in Section 8.1, T_{PERC} adds an annotation to each method call and each occurrence of the **new** operator for indicating the memory area in which the newly created object which is returned by the call (initialised by the constructor) is to be allocated. This annotation is chosen from the following alternatives:

- the `@<oma>` pointer in case the method/constructor call will perform allocations in the caller's (of the enclosing method) scope.
- the expression `@this.<rs>` in case of a method/constructor call allocating an object in the reentrant scope.
- the method invocation `@ImmortalMemory.instance()` in case of a method/constructor invocation allocating an object in immortal memory.
- the `@<coma>` pointer in case the method/constructor call will perform allocations in the constructed scope.
- the `@<cma>` if none of the preceding alternatives applies.

PERC Pico

```
public Foo getFoo(){
  Object tmp = new@<cma> Object();
  Bar bar = new@<coma> Bar();
  Foo foo = new@<oma> Foo();
  foo.bar = bar;
  return foo;
}
```

PERC Pico

Figure 15.2: PERC Pico Code after Application of T_{Perc}

Figure 15.2 shows the result of T_{Perc} being applied to the method `getFoo` introduced in figure 15.1.

The initialisation of the `<coma>` pointer is performed when expanding a method body statement. For this we have to distinguish whether the corresponding method is annotated with **`@ExternallyConstructedScope`** or not.

[2] These analysis results can, for instance, be obtained from the *C* Code generated by PERC Pico as an intermediate stage when compiling Java code.

percECSMemoryMethodBodyExpandMC

$$\begin{array}{l}
\quad \Gamma,\ \{\mathcal{U}\}(cma.\mathtt{stack} \preceq get_{MemoryStack}(j) \land j \geq MemoryStack.\texttt{<ntc>}) \Rightarrow \\
\qquad \{\mathcal{U}\}\{get_{LTMemory}(LTMemory.\texttt{<ntc>}).\mathtt{stack} := get_{LTMemory}(j) \mathrel{||} \\
\qquad\quad newLocalMem := get_{LTMemory}(LTMemory.\texttt{<ntc>}) \mathrel{||} \\
\qquad\quad get_{LTMemory}(LTMemory.\texttt{<ntc>}).\texttt{<c>} := TRUE \mathrel{||} \\
\qquad\quad get_{LTMemory}(LTMemory.\texttt{<ntc>}).consumed := 0 \mathrel{||} \\
\qquad\quad get_{LTMemory}(LTMemory.\texttt{<ntc>}).size := localScopeSize \mathrel{||} \\
\qquad\quad LTMemory.\texttt{<ntc>} := LTMemory.\texttt{<ntc>} + 1 \mathrel{||} \\
\qquad\quad MemoryStack.\texttt{<ntc>} := j + 1 \mathrel{||} \\
\qquad\quad for\ i;\ if(MemoryStack.\texttt{<ntc>} \leq i \land i \leq j) \\
\qquad\qquad\qquad\qquad get_{MemoryStack}(i).\texttt{<c>} := TRUE \\
\qquad\quad \}\langle \pi\ \texttt{method-frame(}\ \texttt{result->lhs,} \\
\qquad\qquad\qquad\qquad\quad \texttt{source=C,} \\
\qquad\qquad\qquad\qquad\quad \mathbf{this}\texttt{=se,} \\
\qquad\qquad\qquad\qquad\quad \texttt{<cma>=newLocalMem,} \\
\qquad\qquad\qquad\qquad\quad \texttt{<oma>=mem,} \\
\qquad\qquad\qquad\qquad\quad \mathbf{<coma>=\ mem})\ :\ \{\texttt{body}\}\ \omega\rangle\phi,\ \Delta \\
\hline
\qquad\qquad \Gamma \Rightarrow \{\mathcal{U}\}\langle \pi\ \texttt{lhs=se.meth(args)@scope@(C);}\ \omega\rangle\ \phi,\ \Delta
\end{array}$$

Figure 15.3: Expanding of a method body statement for a method annotated with the **@ExternallyConstructedScope** annotation

15.2.1 Externally Constructed Scopes

If the constructed scope is provided externally (by the caller) it is identical to the scope holding the returned object which is determined by the annotation @scope attached to the *method body statement*. Figure 15.3 shows the rule for expanding such a method body statement. The changes in comparison to rule scopedMemoryMethodBodyExpandMC are printed bold.

15.2.2 "Internally" (Newly) Constructed Scopes

If a method is not annotated with **@ExternallyConstructedScope** it has to create its own constructed scope[3]. Consequently the <coma> pointer is set to a fresh instance of LTMemory whose size is set as specified by the **@ConstructedScopedMemorySize** annotation. The constructed scope resides on the same scope level as the scope containing the returned object. Here the decoupling of the nesting order (which is defined on scope sub-stacks not on the memory areas themselves) from the actual scopes again pays off: We can simply initialise the stack attribute of the constructed scope with the stack of the outer scope in which the returned object is to be allocated. Thus these two scopes share the same local scope stack but are physically different. This entails that rule stackInjective shown in Section 6.4 is not sound anymore for treating PERC Pico programs (and can consequently not be used for this).

The rule for expanding method bodies creating a fresh constructed scope is shown in figure 15.4. The changes in comparison to rule scopedMemoryMethodBodyExpandMC are again printed bold. We obtain term *consScopeSize* from the **@ConstructedScopedMemorySize**

[3] Also in cases the constructed scope size is specified to be 0 we still assume that a constructed scope of this size is created.

annotation defining the size of the constructed scope.

As already elaborated, a constructed scope is under certain circumstances embedded in the local scope or the reentrant scope of the enclosing method. This is taken into account by update $\mathcal{V}$ increasing the `consumed@(MemoryArea)` attribute of the affected local/reentrant scope accordingly if needed:

- $\{\mathcal{V}\} = \{cma.consumed := cma.consumed + consScopeSize'\}$

  where $consScopeSize' := consScopeSize + LTMemoryArea.size$. Update $\mathcal{V}$ is chosen as above if the object returned by `meth` is to be allocated in the current local scope (`@scope=@<cma>`) and the innermost method frame occurring in π belongs to a method annotated with **`@CallerAllocatedResult`** (resp. **`@CallerAllocatedArrayResult`**) or to the implicit method `<init>` (representing basically a constructor body). The term *cma* stands for the local scope referenced by the `<cma>` pointer of this method frame. The term *LTMemoryArea.size* represents the memory overhead needed (in addition to the space for objects allocated in this scope) for the constructed scope itself whereas *consScopeSize* is the size of the constructed scope available to objects allocated inside it.

- $\{\mathcal{V}\} = \{self.\texttt{<rs>}.consumed := self.\texttt{<rs>}.consumed + consScopeSize'\}$

  If `@scope=@this.<rs>`. The term *self* stands for the receiver object referenced by the **`this`** pointer of the innermost enclosing method frame occurring in π.

- $\{\mathcal{V}\} = \{skip\}$ if none of the preceding cases matches.

15.2.3 Initialisation of the Implicit Field `<rs>`

When invoking the constructor declared in a class annotated with **`@ReentrantScope`**, the implicit field `<rs>` is initialised with the constructed scope associated with this constructor. If **`this.`**`<rs>` has already been initialised by a chained **`this`** or **`super`** constructor invocation with a constructed scope different from the current constructed scope[4] (i.e., the one `<coma>` refers to) the current constructed scope is resized accordingly (see Section 15.1.4). Technically this is realized by placing the code fragment[5]

— Java —

```
if(this.<rs>!=<coma>) {
  if(this.<rs>!=null) {
    <coma>.size += this.<rs>.size;
    <coma>.consumed += this.<rs>.consumed;
  }
  this.<rs> = <coma>;
}
```

— Java —

[4] This occurs if the declaration of the invoked **`this`** or **`super`** constructor is *not* annotated with **`@ExternallyConstructedScope`**.

[5] The field **`this.`**`<rs>` has already been set to **`null`** when allocating the regarded object (and thus before executing any implementation of the `<init>` method on the regarded object). For the sake of brevity we omit the canonical changes applied to the allocation rule allocateMIB for realizing this.

percCSMemoryMethodBodyExpandMC

$$
\begin{array}{l}
\quad \Gamma,\ \{\mathcal{U}\}(cma.\mathtt{stack} \preceq get_{MemoryStack}(j) \land j \geq MemoryStack.<\mathtt{ntc}>) \Rightarrow \\
\qquad \{\mathcal{U}\}\{\boldsymbol{\mathcal{V}}\}\{get_{LTMemory}(LTMemory.<\mathtt{ntc}>).\mathtt{stack} := get_{LTMemory}(j)\ || \\
\qquad\qquad\qquad newLocalMem := get_{LTMemory}(LTMemory.<\mathtt{ntc}>)\ || \\
\qquad\qquad\qquad get_{LTMemory}(LTMemory.<\mathtt{ntc}>).<\mathtt{c}> := TRUE\ || \\
\qquad\qquad\qquad get_{LTMemory}(LTMemory.<\mathtt{ntc}>).consumed := 0\ || \\
\qquad\qquad\qquad get_{LTMemory}(LTMemory.<\mathtt{ntc}>).size := localScopeSize\ || \\
\qquad\qquad\qquad \mathbf{newConsMem} := \mathbf{get_{LTMemory}}(\mathbf{LTMemory}.<\mathbf{ntc}> + \mathbf{1})\ || \\
\qquad\qquad\qquad \mathbf{get_{LTMemory}}(\mathbf{LTMemory}.<\mathbf{ntc}> + \mathbf{1}).\mathbf{stack} := \mathbf{mem}.\mathbf{stack}\ || \\
\qquad\qquad\qquad \mathbf{get_{LTMemory}}(\mathbf{LTMemory}.<\mathbf{ntc}> + \mathbf{1}).<\mathbf{c}> := \mathbf{TRUE}\ || \\
\qquad\qquad\qquad \mathbf{get_{LTMemory}}(\mathbf{LTMemory}.<\mathbf{ntc}> + \mathbf{1}).\mathbf{consumed} := \mathbf{0}\ || \\
\qquad\qquad\qquad \mathbf{get_{LTMemory}}(\mathbf{LTMemory}.<\mathbf{ntc}> + \mathbf{1}).\mathbf{size} := \mathbf{consScopeSize}\ || \\
\qquad\qquad\qquad LTMemory.<\mathtt{ntc}> := LTMemory.<\mathtt{ntc}> + 2\ || \\
\qquad\qquad\qquad MemoryStack.<\mathtt{ntc}> := j + 1\ || \\
\qquad\qquad\qquad for\ i;\ if(MemoryStack.<\mathtt{ntc}> \leq i \land i \leq j) \\
\qquad\qquad\qquad\qquad\qquad\qquad get_{MemoryStack}(i).<\mathtt{c}> := TRUE \\
\qquad\qquad\qquad \}\langle \pi\ \mathtt{method\text{-}frame(}\ \mathtt{result\text{-}>lhs,} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{source=C,} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{this}\mathtt{=se,} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{<cma>=newLocalMem,} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{<oma>=mem,} \\
\qquad\qquad\qquad\qquad\qquad\qquad <\mathbf{coma}>= \mathbf{newConsMem})\ :\ \{\mathtt{body}\}\ \omega\rangle\phi,\ \Delta \\
\hline
\qquad\qquad\qquad\qquad \Gamma \Rightarrow \{\mathcal{U}\}\langle\pi\ \mathtt{lhs=se.meth(args)@scope@(C);}\ \omega\rangle\ \phi,\ \Delta
\end{array}
$$

Figure 15.4: Expanding of a method body statement of methods *not* declared with the **`@ExternallyConstructedScope`** annotation

in the `<init>` method right after the possible call to a **`this`** or **`super`** constructor. For details on the structure of the `<init>` method refer to [Beckert et al., 2007, p. 143].

15.3 Proof Obligation for PERC Pico Programs

The proof obligation for proving the correctness (i.e., sufficiency) of *constructed* scope sizes can be defined as follows:

$$\begin{aligned}&RespectsConstructedScopeSize(ct; Assumed) :=\\&\quad Conj_{Assumed} \wedge Pre \wedge Conj_{InvRealtime} \wedge ValidCall_{op}^{SCJ} \wedge ValidCall_{op}^{PERC} \rightarrow\\&\quad \{LTMemory.\texttt{<ntc>}^{@pre} := LTMemory.\texttt{<ntc>}\}\langle Prg_{op}()\rangle\, cs.consumed \leq cs.size\end{aligned}$$

where

- *cs* abbreviates
  - $defaultMemoryArea$ if *op* is annotated with **`@ExternallyConstructedScope`**
  - $get_{LTMemory}(LTMemory.\texttt{<ntc>}^{@pre} + 1)$ otherwise

  which is the constructed scope of the method call embedded in $Prg_{op}()$,
- $Conj_{Assumed}$ is defined as in Section 2.3.5,
- Pre is defined as in Section 2.3.5,
- $Conj_{InvRealtime}$ is defined as in Section 7.2,
- The static analysis performed by PERC Pico enforces certain constraints on arguments of methods not declared with the **`@AllowCheckedScopedLinks`** modifier. These constraints are expressed by the formula $ValidCall_{op}^{PERC}$ in the above PO. Let $p_1, \ldots, p_n$ be the arguments of the method call embedded in $Prg_{op}()$
  - For a method declared with the **`@CallerAllocatedResult`** modifier all arguments of the method declared as **`@Scoped`** (but not **`@CaptiveScoped`**[6]) are known to reside in an outer scope of the constructed scope or (in the case of an externally constructed scope) in the constructed scope itself.
  - For a method declared with the **`@ReentrantScope`** modifier or a constructor all arguments declared as **`@Scoped`** (but not **`@CaptiveScoped`**) reside in the same scope as the **`this`** object or an enclosing scope of it.

  In both of the above cases $ValidCall_{op}^{PERC}$ has the form

$$ValidCall_{op}^{PERC} := \bigwedge_{\substack{i:1\leq i\leq n \text{ and}\\ p_i \text{ is declared } \texttt{@Scoped}}} (p_i \not\doteq \textbf{null} \rightarrow p_i.\texttt{<ma>}.\texttt{stack} \preceq mem.\texttt{stack}) \qquad (15.1)$$

  where *mem* denotes either the (externally) constructed scope or the scope containing the receiver object (created object) of the method (constructor) invocation embedded in

[6] The modifier **`@CaptiveScoped`** indicates that no reference to the annotated parameter created by the method execution outlives the method execution.

$Prg_{op}()$. For methods and constructors annotated with the **`@NestedReentrantScope`** modifier restrictions on **`@Scoped`** arguments are slightly more stringent: **`@Scoped`** arguments are required to reside in the same scope as the receiver object of the method (or the object created by the constructor) itself. This leads to the following definition of $ValidCall_{op}^{PERC}$ for this kind of methods and constructors:

$$ValidCall_{op}^{PERC} := \bigwedge_{\substack{i:1\leq i\leq n \text{ and} \\ p_i \text{ is declared } \texttt{@Scoped}}} (p_i \not\doteq \texttt{null} \rightarrow p_i.\texttt{<ma>} \doteq mem) \qquad (15.2)$$

where mem is the memory area containing the object referenced by *op*'s **`this`** pointer.

Proof obligations for the sufficiency of local and reentrant scopes are defined analogously.

15.4 Modular Verification of PERC Pico Programs

This section lists modifications needed to apply the calculus presented in 13 to PERC Pico programs. These modifications are mostly canonical and their implementation is not described in its technical details here.

15.4.1 Memory Consumption in the Caller Context

PERC Pico annotations lack a facility to specify the amount of memory allocated in the caller context. This is, however, needed for performing modular verification in cases externally constructed scopes or returned objects of an array type are involved (In case a method returns a non-array object of type T, externally constructed scopes are not allowed to be used and the constructed scope is also not embedded in the outer local scope we can assume that the amount of memory allocated in the caller context does not exceed $T.$`<size>`).

Rules for symbolically executing method invocations by approximating them by their contract need (analogously to rules for expanding method bodies) to determine the scope for allocating a newly created returned object in. The memory space consumed in this and other already existing scopes has to be accounted for (and needs to be specified by a contract).

15.4.2 Memory Consumption in Loops

Since, with *local*, *constructed* and *reentrant* scopes, we now have to deal with several *currently active* scopes in parallel, it is no longer sufficient to have only one *working space* clause for specifying the memory consumption (in the current scope) of a single loop iterations. Memory consumption figures in the different current scopes need to be distinguished by different loop-level working space clauses. The loop invariant rules presented in Section 13.2.3 are adapted canonically.

16 Related Work

The shortcomings of JML's `\working_space` constructs were also identified in [Atkey, 2006]. To overcome this, [Atkey, 2006] proposes to specify the working space of certain methods with the help of model methods parameterized by those locations and method arguments potentially affecting the working space of the specified method. This carries the advantage of requiring only existing JML features but burdens the programmer with extra specification efforts (providing and specifying the model method) and decreases modularity (changes in the program might entail changes of model methods and specifications containing calls to the changed model methods).

In [Barthe et al., 2005] another approach verifying the JML memory consumption contracts is presented. The total amount of consumed heap space is stored in a ghost variable which is increased by **set** statements after each constructor call. The memory specifications described in [Barthe et al., 2005] can in principle be verified modularly but not conveniently be specified in a modular way since the memory consumption of a method cannot be expressed (the way memory contracts are written in [Barthe et al., 2005]) as a function of the memory consumption of the methods it calls (as it is possible with the `\working_space` construct). Using the memory contracts introduced in [Barthe et al., 2005] this could, however, still be achieved by means of model methods as described in [Atkey, 2006].

The verification of loop bounds is described in [Hunt et al., 2006]. Section 13.2.3 extends and makes use of this work by describing a way to reason about the memory consumption of loops.

Giambiagi and Schneider [Giambiagi and Schneider, 2005] presented an on-card static analyzer for estimating the memory consumption of Java Card byte code. A similar approach is described in [Pham et al., 2008]. Performing a sound analysis automatically on a smart card with very limited resources implicates of course a preciseness trade-off: The algorithm presented in [Giambiagi and Schneider, 2005], for instance, basically is limited to detecting whether a **new** instruction occurs inside a cycle (i.e., a loop or recursively called method) and, in case such a **new** instruction is found, assumes infinitely many instances to be allocated by this instruction.

There are several approaches, as, for instance, [Hughes et al., 1996, Chin et al., 2005], to make space requirements of objects part of the type system.

Part IV

Conclusions

17 Summary and Future Work

This work provided several contributions in the field of formal verification of safety-critical and real-time JAVA systems some of which are also relevant and applicable to JAVA CARD (such as the approach for verifying memory performance contracts, as described in Sections 10–14) and standard JAVA applications (such as the proof obligations discussed in Section 9).

The first part of this thesis laid the foundations the remainder of this work is based on. Section 2 gave an overview of JAVA DL and introduced solutions for handling inner classes and sum comprehensions in the JAVA DL calculus which formed a prerequisite for other parts of this work. For treating sum comprehensions we assumed two axioms, the correctness (relative to these axioms) of all other rules introduced for treating comprehensions was verified with the KeY system. Some evidence was provided suggesting a high degree of automation facilitated by this rule set when employed in induction proofs. The subsequent Sections 3 and 4 summed up the aspects of JML and RTSJ relevant for this work.

The second part deals with the formalisation of the RTSJ and other region-based memory models and implications of these memory models for proving data encapsulation properties. Section 6 showed how the RTSJ MM can be formalized in JAVA DL. For this we imposed some restrictions on the set of considered programs as described in Section 5. Proof obligations for RTSJ programs were considered in Section 7 which also exemplarily showed how higher level specification means can be defined based on the constructs introduced in the preceding Sections. In the following (Section 8) adaptations of the presented approach required for treating other SCJ profiles were discussed. Section 9 reviews several proof obligations for proving data encapsulation properties, introduces novel POs for *depends* and *captures* clauses and provides correctness and completeness proofs for these POs. It is also explained how the RTSJ MM can be leveraged for facilitating non-interference in RTSJ programs.

The third part depicted a contract-based approach for verifying worst case memory usage (WCMU) specifications. Section 11 wrapped up the existing JML specification means for WCMU figures, pointed out their shortcomings and made propositions how to overcome them. In the following, a calculus for verifying the correctness of WCMU contracts in a modular way was introduced. This included a loop invariant rule for verifying the memory consumption of loops. Adaptations of this calculus required for treating memory models featuring implicit scope identities were developed in Section 15. Particular focus was put on the memory model employed by the PERC Pico tool suite [Nilsen, 2006].

With few exceptions the concepts described in this work have been implemented in the KeY system.

Possible areas of future work are the evaluation of the presented solutions based on a wider range of examples and the minimisation of user interaction: Currently all specifications described in this work have to be provided by the user. It would be preferable to derive some of them, such as, for instance, the memory consumption figures of single loop iterations or loop invariants [Schmitt and Weiß, 2007], automatically by means of static analysis. More lightweight static analysis techniques (such as abstract interpretation [Cousot and Cousot, 1977]) could also help to provide likely memory consumption estimations. The applied analysis could attempt to be as precise as possible without being restricted by the necessity of being sound since its results can then be checked by means of formal verification as described in this work. Another topic worth investigating would be whether the verification of *depends* and *captures* clauses as presented in Section 9 would benefit from a different, more explicit encoding of the heap which could simplify formulating constraints on heaps or comparing two different heaps.

A Taclets for *Sum* Comprehensions

This Appendix lists all rules for treating *sum* comprehensions (introduced in Section 2.3.8).

A.1 Not Automatically Applicable Rules

$$\textsf{sumIndU1}\quad \mathrm{sum}(x;\ [l,u);\ t) \rightsquigarrow \mathrm{sum}(x;\ [l,u-1);\ t) + \backslash if(l<u)\,\backslash then(t^{[x/u-1]})\,\backslash else(0)$$

$$\textsf{sumIndU2}\quad \mathrm{sum}(x;\ [l,u);\ t) \rightsquigarrow \mathrm{sum}(x;\ [l,u+1);\ t) - \backslash if(l<u+1)\,\backslash then(t^{[x/u]})\,\backslash else(0)$$

$$\textsf{sumIndL1}\quad \mathrm{sum}(x;\ [l,u);\ t) \rightsquigarrow \mathrm{sum}(x;\ [l+1,u);\ t) + \backslash if(l<u)\,\backslash then(t^{[x/l]})\,\backslash else(0)$$

$$\textsf{sumIndL2}\quad \mathrm{sum}(x;\ [l,u);\ t) \rightsquigarrow \mathrm{sum}(x;\ [l-1,u);\ t) - \backslash if(l-1<u)\,\backslash then(t^{[x/l-1]})\,\backslash else(0)$$

$$\textsf{sumDistributive}\quad \mathrm{sum}(x;\ [l,u);\ t_1 * t_2) \rightsquigarrow \mathrm{sum}(x;\ [l,u);\ t_1) * t_2$$

Where x has no free occurrence in t_2.

$$\textsf{sumSplit}\ \dfrac{\begin{array}{l}\Gamma \Rightarrow l \leq m \wedge m \leq u,\ \Delta\\ [\mathrm{sum}(x;\ [l,u);\ t) \rightsquigarrow \mathrm{sum}(x;\ [l,m);\ t) + \mathrm{sum}(x;\ [m,u);\ t)]\end{array}}{\Gamma \Rightarrow \Delta}\ \ni \mathrm{sum}(x;\ [l,u);\ t)$$

$$\textsf{sumEmpty}\ \dfrac{\begin{array}{c}\Gamma \Rightarrow u \leq l,\ \Delta\\ [\mathrm{sum}(x;\ [l,u);\ t) \rightsquigarrow 0]\end{array}}{\Gamma \Rightarrow \Delta}\ \ni \mathrm{sum}(x;\ [l,u);\ t)$$

$$\textsf{singleSummand}\ \dfrac{\begin{array}{c}\Gamma \Rightarrow l \doteq u-1,\ \Delta\\ \left[\mathrm{sum}(x;\ [l,u);\ t) \rightsquigarrow t^{[x/l]}\right]\end{array}}{\Gamma \Rightarrow \Delta}\ \ni \mathrm{sum}(x;\ [l,u);\ t)$$

A.2 Automatically Applicable Rules

$$\textsf{sumIndU1Concr}\quad \mathrm{sum}(x;\ [l,1+u);\ t) \rightsquigarrow \mathrm{sum}(x;\ [l,u);\ t) + \backslash if(l \leq u)\,\backslash then(t^{[x/u]})\,\backslash else(0)$$

$$\textsf{sumIndU2Concr}\quad \mathrm{sum}(x;\ [l,-1+u);\ t) \rightsquigarrow \mathrm{sum}(x;\ [l,u);\ t) - \backslash if(l<u)\,\backslash then(t^{[x/u-1]})\,\backslash else(0)$$

$$\textsf{sumIndL1Concr}\quad \mathrm{sum}(x;\ [l-1,u);\ t) \rightsquigarrow \mathrm{sum}(x;\ [l,u);\ t) + \backslash if(l<u)\,\backslash then(t^{[x/l-1]})\,\backslash else(0)$$

$$\textsf{sumIndL2Concr}\quad \mathrm{sum}(x;\ [1+l,u);\ t) \rightsquigarrow \mathrm{sum}(x;\ [l,u);\ t) - \backslash if(l<u)\,\backslash then(t^{[x/l]})\,\backslash else(0)$$

$$\textsf{sumEmptyConcrete1}\ \dfrac{\Gamma \Rightarrow \Delta,\ u \leq l}{\Gamma \Rightarrow \mathrm{sum}(x;\ [l,u);\ t) \doteq 0,\ \Delta}$$

$$\textsf{sumEmptyConcrete2}\quad \mathrm{sum}(x;\ [l,-u);\ t) \rightsquigarrow 0$$

where l and u are positive integer literals.

$$\textsf{sumLEqU}\quad \mathrm{sum}(x;\ [l,l);\ t) \rightsquigarrow 0$$

$$\mathsf{sumOneZero}\ \frac{\Gamma,\ s \geq 0 \Rightarrow \Delta}{\Gamma \Rightarrow \Delta} \ni s$$

where s stands for a term of the form $\mathrm{sum}(x;\ [l,u);\ \backslash \mathit{if}(b)\,\backslash \mathit{then}(1)\,\backslash \mathit{else}(0))$.

$$\mathsf{sumEqual1}\ \frac{\Gamma \Rightarrow \mathrm{sum}(x;\ [l,u);\ t_1) \doteq \mathrm{sum}(y;\ [l,u);\ t_2),\ \forall x;\ l \leq x \wedge x < u \rightarrow t_1 \doteq t_2^{y/x},\ \Delta}{\Gamma \Rightarrow \mathrm{sum}(x;\ [l,u);\ t_1) \doteq \mathrm{sum}(y;\ [l,u);\ t_2),\ \Delta}$$

$$\mathsf{sumCommutativeAssociative}\quad \mathrm{sum}(x;\ [l,u);\ t_1 + t_2) \rightsquigarrow \mathrm{sum}(x;\ [l,u);\ t_1) + \mathrm{sum}(x;\ [l,u);\ t_2)$$

$$\mathsf{sumEqual2}\ \frac{\begin{array}{c}\Gamma,\ \mathrm{sum}(x;\ [l,u);\ t_1) \doteq t \Rightarrow \\ \mathrm{sum}(y;\ [l,u);\ t_2) \doteq t,\ \forall x;\ l \leq x \wedge x < u \rightarrow t_1 \doteq t_2^{y/x},\ \Delta\end{array}}{\Gamma,\ \mathrm{sum}(x;\ [l,u);\ t_1) \doteq t \Rightarrow \mathrm{sum}(y;\ [l,u);\ t_2) \doteq t,\ \Delta}$$

$$\mathsf{sumEqual3}\ \frac{\begin{array}{c}\Gamma,\ \mathrm{sum}(x;\ [l,u);\ t_1) \doteq i,\ \mathrm{sum}(y;\ [l,u);\ t_2) \doteq j \Rightarrow \\ i \doteq j,\ \forall x;\ l \leq x \wedge x < u \rightarrow t_1 \doteq t_2^{y/x},\ \Delta\end{array}}{\Gamma,\ \mathrm{sum}(x;\ [l,u);\ t_1) \doteq i,\ \mathrm{sum}(y;\ [l,u);\ t_2) \doteq j \Rightarrow i \doteq j,\ \Delta}$$

$$\mathsf{sumEqZeroCut}\ \frac{\begin{array}{c}\Gamma \Rightarrow \mathrm{sum}(x;\ [l_1,u_1);\ t_1) \doteq 0,\ \mathrm{sum}(x;\ [l_1,u_1);\ t_1) \doteq \mathrm{sum}(y;\ [l_2,u_2);\ t_2) * t,\ \Delta \\ \Gamma,\ \mathrm{sum}(x;\ [l_1,u_1);\ t_1) \doteq 0 \Rightarrow \mathrm{sum}(x;\ [l_1,u_1);\ t_1) \doteq \mathrm{sum}(y;\ [l_2,u_2);\ t_2) * t,\ \Delta\end{array}}{\Gamma \Rightarrow \mathrm{sum}(x;\ [l_1,u_1);\ t_1) \doteq \mathrm{sum}(y;\ [l_2,u_2);\ t_2) * t,\ \Delta}$$

$$\mathsf{sumZeroRight}\ \frac{\Gamma \Rightarrow \forall\ \mathit{int}\ x; l \leq x \wedge x < u \rightarrow t \doteq 0,\ \mathrm{sum}(x;\ [l,u);\ t) \doteq 0,\ \Delta}{\Gamma \Rightarrow \mathrm{sum}(x;\ [l,u);\ t) \doteq 0,\ \Delta}$$

$$\mathsf{sumEqSplit1}\ \frac{\begin{array}{ll}\Gamma \Rightarrow & \begin{array}{l} l \leq u_1 \wedge l \leq u_2 \wedge \backslash \mathit{if}(u_1 < u_2) \\ \qquad \backslash \mathit{then}(\mathrm{sum}(x;\ [l,u_1);\ t_1 - t_2^{[y/x]}) \doteq \mathrm{sum}(y;\ [u_1,u_2);\ t_2)) \\ \qquad \backslash \mathit{else}(\mathrm{sum}(x;\ [u_2,u_1);\ t_1) \doteq \mathrm{sum}(y;\ [l,u_2);\ t_2 - t_1^{[x/y]})), \\ \mathrm{sum}(x;\ [l,u_1);\ t_1) \doteq \mathrm{sum}(y;\ [l,u_2);\ t_2),\ \Delta \end{array}\end{array}}{\Gamma \Rightarrow \mathrm{sum}(x;\ [l,u_1);\ t_1) \doteq \mathrm{sum}(y;\ [l,u_2);\ t_2),\ \Delta}$$

$$\mathsf{sumEqSplit1'}\ \frac{\begin{array}{ll}\Gamma \Rightarrow & \begin{array}{l} l \leq u_1 \wedge l \leq u_2 \wedge \backslash \mathit{if}(u_1 < u_2) \\ \qquad \backslash \mathit{then}(\mathrm{sum}(x;\ [l,u_1);\ t_1 - t_2^{[y/x]}) \doteq \mathrm{sum}(y;\ [u_1,u_2);\ t_2)) \\ \qquad \backslash \mathit{else}(\mathrm{sum}(x;\ [u_2,u_1);\ t_1) \doteq \mathrm{sum}(y;\ [l,u_2);\ t_2 - t_1^{[x/y]})), \\ \mathrm{sum}(x;\ [l,u_1);\ t_1) \doteq \mathrm{sum}(y;\ [l,u_2);\ t_2),\ \Delta \end{array}\end{array}}{\Gamma,\ \mathrm{sum}(x;\ [l,u_1);\ t_1) \doteq t \Rightarrow \mathrm{sum}(y;\ [l,u_2);\ t_2) \doteq t,\ \Delta}$$

$$\mathsf{sumEqSplit2}\ \frac{\begin{array}{ll}\Gamma \Rightarrow & \begin{array}{l} l_1 \leq u \wedge l_2 \leq u \wedge \backslash \mathit{if}(l_1 < l_2) \\ \qquad \backslash \mathit{then}(\mathrm{sum}(x;\ [l_1,l_2);\ t_1) \doteq \mathrm{sum}(y;\ [l_2,u);\ t_2 - t_1^{x/y})) \\ \qquad \backslash \mathit{else}(\mathrm{sum}(x;\ [l_1,u);\ t_1 - t_2^{y/x}) \doteq \mathrm{sum}(y;\ [l_2,l_1);\ t_2)), \\ \mathrm{sum}(x;\ [l,u_1);\ t_1) \doteq \mathrm{sum}(y;\ [l,u_2);\ t_2),\ \Delta \end{array}\end{array}}{\Gamma \Rightarrow \mathrm{sum}(x;\ [l_1,u);\ t_1) \doteq \mathrm{sum}(y;\ [l_2,u);\ t_2),\ \Delta}$$

$$\mathsf{sumEqSplit2'}\ \frac{\begin{array}{ll}\Gamma \Rightarrow & \begin{array}{l} l_1 \leq u \wedge l_2 \leq u \wedge \backslash \mathit{if}(l_1 < l_2) \\ \qquad \backslash \mathit{then}(\mathrm{sum}(x;\ [l_1,l_2);\ t_1) \doteq \mathrm{sum}(y;\ [l_2,u);\ t_2 - t_1^{x/y})) \\ \qquad \backslash \mathit{else}(\mathrm{sum}(x;\ [l_1,u);\ t_1 - t_2^{y/x}) \doteq \mathrm{sum}(y;\ [l_2,l_1);\ t_2)), \\ \mathrm{sum}(x;\ [l,u_1);\ t_1) \doteq \mathrm{sum}(y;\ [l,u_2);\ t_2),\ \Delta \end{array}\end{array}}{\Gamma,\ \mathrm{sum}(x;\ [l_1,u);\ t_1) \doteq t \Rightarrow \mathrm{sum}(y;\ [l_2,u);\ t_2) \doteq t,\ \Delta}$$

Bibliography

[Andreae et al., 2007] Andreae, C., Coady, Y., Gibbs, C., Noble, J., Vitek, J., and Zhao, T. (2007). Scoped types and aspects for real-time Java memory management. *Real-Time Syst.*, 37(1):1–44.

[Aonix, 2008] Aonix (2008). *PERC Pico User Manual*. `http://research.aonix.com/jsc/`.

[Atkey, 2006] Atkey, R. (2006). Specifying and verifying heap space allocation with JML and ESC/Java2 (preliminary report). In *Workshop on Formal Techniques for Java-like Programs (FTfJP)*.

[Barnett et al., 2005] Barnett, M., Leino, and Schulte, W. (2005). *The Spec# Programming System: An Overview*, volume 3362/2005 of *Lecture Notes in Computer Science*, pages 49–69. Springer, Berlin / Heidelberg.

[Barthe et al., 2005] Barthe, G., Pavlova, M., and Schneider, G. (2005). Precise analysis of memory consumption using program logics. *SEFM*, 0:86–95.

[Baum, 2007] Baum, M. (2007). Debugging by visualizing symbolic execution. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe.

[Beckert et al., 2004] Beckert, B., Giese, M., Habermalz, E., Hähnle, R., Roth, A., Rümmer, P., and Schlager, S. (2004). 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). Special Issue on Symbolic Computation in Logic and Artificial Intelligence.

[Beckert and Gladisch, 2007] Beckert, B. and Gladisch, C. (2007). White-box testing by combining deduction-based specification extraction and black-box testing. In Meyer, B. and Gurevich, Y., editors, *Proceedings, International Conference on Tests and Proofs (TAP), Zurich, Switzerland*, LNCS 4454. Springer.

[Beckert et al., 2007] Beckert, B., Hähnle, R., and Schmitt, P. H., editors (2007). *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer.

[Beckert et al., 2002] Beckert, B., Keller, U., and Schmitt, P. H. (2002). Translating the Object Constraint Language into first-order predicate logic. In *Proceedings, VERIFY, Workshop at Federated Logic Conferences (FLoC), Copenhagen, Denmark*.

[Beckert and Klebanov, 2007] Beckert, B. and Klebanov, V. (2007). A dynamic logic for deductive verification of concurrent programs. In Hinchey, M. and Margaria, T., editors, *Proceedings, 5th IEEE International Conference on Software Engineering and Formal Methods (SEFM), London, UK*. IEEE Press.

[Beebee and Rinard, 2001] Beebee, W. S. and Rinard, M. C. (2001). An implementation of scoped memory for real-time java. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 289–305, London, UK. Springer-Verlag.

[Bengtsson et al., 1996] Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., and Yi, W. (1996). Uppaal—a tool suite for automatic verification of real-time systems. In *Proceedings of the DIMACS/SYCON workshop on Hybrid systems III : verification and control*, pages 232–243, Secaucus, NJ, USA. Springer-Verlag New York, Inc.

[Bollella and Gosling, 2000] Bollella, G. and Gosling, J. (2000). The real-time specification for Java. *Computer*, 33(6):47–54.

[Boyapati et al., 2003] Boyapati, C., Salcianu, A., Beebee, W., and Rinard, J. (2003). Ownership types for safe region-based memory management in real-time Java. *ACM Conference on Programming Language Design and Implementation (PLDI).*

[Breunesse and Poll, 2003] Breunesse, C. and Poll, E. (2003). Verifying jml specifications with model fields. Technical report, In Formal Techniques for Java-like Programs. Proceedings of the ECOOP 2003 Workshop.

[Bubel, 2007] Bubel, R. (2007). *Formal verification of recursive predicates.* PhD thesis, Universität Karlsruhe.

[Bubel et al., 2008a] Bubel, R., Hähnle, R., and Schmitt, P. H. (2008a). Specification predicates with explicit dependency information. In Beckert, B., editor, *Proceedings, 5th International Verification Workshop (VERIFY'08)*, volume 372 of *CEUR Workshop Proceedings*, pages 28–43. CEUR-WS.org.

[Bubel et al., 2008b] Bubel, R., Roth, A., and Rümmer, P. (2008b). Ensuring the correctness of lightweight tactics for javacard dynamic logic. *Electron. Notes Theor. Comput. Sci.*, 199:107–128.

[Chen, 2000] Chen, Z. (2000). *Java Card Technology for Smart Cards: Architecture and Programmer's Guide.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Chin et al., 2005] Chin, W., Nguyen, H. H., Qin, S., and Rinard, M. (2005). Memory usage verification for oo programs. In *In SAS 05*, pages 70–86. Springer.

[Choi et al., 1999] Choi, J.-D., Gupta, M., Serrano, M., Sreedhar, V. C., and Midkiff, S. (1999). Escape analysis for Java. *SIGPLAN Not.*, 34(10):1–19.

[Cok and Kiniry, 2004] Cok, D. R. and Kiniry, J. (2004). Esc/java2: Uniting esc/java and jml. In *CASSIS*, pages 108–128.

[Cousot and Cousot, 1977] Cousot, P. and Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA. ACM.

[Dautelle, 2009] Dautelle, J.-M. (2009). Javolution – the java solution for real-time and embedded systems. Javolution homepage, `http://www.javolution.org/`.

[Dawson, 2008] Dawson, M. H. (2008). Challenges in Implementing the Real-Time Specification for Java (RTSJ) in a Commercial Real-Time Java Virtual Machine. In *ISORC '08: Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 241–247, Washington, DC, USA. IEEE Computer Society.

[Dijkstra and Feijen, 1988] Dijkstra, E. W. and Feijen, W. H. (1988). *A Method of Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Engel, 2005] Engel, C. (2005). A Translation from JML to JavaDL. Studienarbeit, Fakultät für Informatik, Universität Karlsruhe. Institut für Logik, Komplexität und Deduktionssysteme.

[Engel and Hähnle, 2007] Engel, C. and Hähnle, R. (2007). Generating unit tests from formal proofs. In Gurevich, Y. and Meyer, B., editors, *Proceedings, 1st International Conference on Tests And Proofs (TAP), Zurich, Switzerland*, volume 4454 of *LNCS*, pages 169–188. Springer.

[Engel et al., 2009] Engel, C., Roth, A., Schmitt, P., and Weiß, B. (2009). Verification of Modifies Clauses in Dynamic Logic with Non-rigid Functions. Technical Report 2009,9, ISSN: 1432-7864, Universität Karlsruhe.

[Fredriksson et al., 2007] Fredriksson, J., Nolte, T., Nolin, M., and Schmidt, H. (2007). Contract-based reusable worst-case execution time estimate. In *RTCSA '07: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 39–46, Washington, DC, USA. IEEE Computer Society.

[Giambiagi and Schneider, 2005] Giambiagi, P. and Schneider, G. (2005). Memory consumption analysis of Java smart cards. In *Proceedings of XXXI Latin American Informatics Conference (CLEI 2005)*, page 12, Cali, Colombia.

[Giese, 2004] Giese, M. (2004). Taclets and the KeY prover. In Aspinall, D. and Lüth, C., editors, *Proc. User Interfaces for Theorem Provers Workshop, UITP 2003*, volume 103 of *Electronic Notes in Theoretical Computer Science*, pages 67–79. Elsevier.

[Gosling et al., 2005] Gosling, J., Joy, B., Steele, G., and Bracha, G. (2005). *The Java Language Specification, Third Edition.* Addison-Wesley Longman, Amsterdam, 3. edition.

[Harel, 1984] Harel, D. (1984). Dynamic logic. In Gabbay, D. and Guenther, F., editors, *Handbook of Philosophical Logic*, volume 2 of *Extensions of Classical Logic*, pages 497–604. D. Reidel Publishing Company.

[Hayes and Utting, 2001] Hayes, I. J. and Utting, M. (2001). A sequential real-time refinement calculus. *Acta Informatica*, 37(6):385–448.

[HIJA, 2006] HIJA (2006). HIJA – High Integrity Java Application. Project Web Site. `http://www.hija.info`.

[Holzner, 2004] Holzner, S. (2004). *Eclipse.* O'Reilly, 1. edition.

[Hughes et al., 1996] Hughes, J., Pareto, L., and Sabry, A. (1996). Proving the correctness of reactive systems using sized types. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 410–423, New York, NY, USA. ACM.

[Hunt et al., 2006] Hunt, J. J., Siebert, F. B., Schmitt, P. H., and Tonin, I. (2006). Provably correct loops bounds for realtime Java programs. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 162–169, New York, NY, USA. ACM Press.

[Igarashi and Pierce, 2002] Igarashi, A. and Pierce, B. C. (2002). On inner classes. *Inf. Comput.*, 177(1):56–89.

[Jacobs et al., 2007] Jacobs, B., Muller, P., and Piessens, F. (2007). Sound reasoning about unchecked exceptions. In *SEFM '07: Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007)*, pages 113–122, Washington, DC, USA. IEEE Computer Society.

[Jones and Lins, 1996] Jones, R. and Lins, R. (1996). *Garbage collection: algorithms for automatic dynamic memory management.* John Wiley & Sons, Inc., New York, NY, USA.

[King, 1976] King, J. C. (1976). Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394.

[Klebanov, 2004] Klebanov, V. (2004). A JMM-faithful non-interference calculus for Java. In *Scientific Engineering of Distributed Java Applications, 4th International Workshop, FIDJI 2004, Luxembourg-Kirchberg*, volume 3409 of *Lecture Notes in Computer Science*, pages 101–111. Springer.

[Klebanov et al., 2005] Klebanov, V., Rümmer, P., Schlager, S., and Schmitt, P. H. (2005). Verification of JCSP programs. In Broenink, J., Roebbers, H., Sunter, J., Welch, P., and Wood, D., editors, *Communicating Process Architectures 2005*, volume 63 of *Concurrent Systems Engineering Series*, pages 203–218, IOS Press, The Netherlands. IOS Press.

[Krone et al., 2001] Krone, J., Ogden, W. F., and Sitaraman, M. (2001). Modular verification of performance constraints. In *ACM OOPSLA Workshop on Specification and Verification of Component-Based Systems (SAVCBS)*, pages 60–67.

[Kung et al., 2006] Kung, A., Hunt, J., Gauthier, L., and Richard-Foy, M. (2006). Issues in building an ANRTS platform. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 144–151, New York, NY, USA. ACM.

[Kwon et al., 2005] Kwon, J., Wellings, A., and King, S. (2005). Ravenscar-Java: a high-integrity profile for real-time Java: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(5-6):681–713.

[Kwon and Wellings, 2004] Kwon, J. and Wellings, A. J. (2004). Memory management based on method invocation in RTSJ. In *OTM Workshops*, pages 333–345.

[Leavens et al., 2006] Leavens, G. T., Baker, A. L., and Ruby, C. (2006). Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38.

[Leavens et al., 2007] Leavens, G. T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D. R., Müller, P., Kiniry, J., and Chalin, P. (2007). JML Reference Manual. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`.

[Leino, 1998] Leino, K. R. M. (1998). Data groups: specifying the modification of extended state. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 144–153, New York, NY, USA. ACM.

[Leino and Monahan, 2007] Leino, K. R. M. and Monahan, R. (2007). Automatic verification of textbook programs that use comprehensions. In *Workshop on Formal Techniques for Java-like Programs (FTfJP)*.

[Mann et al., 2005] Mann, T., Deters, M., LeGrand, R., and Cytron, R. K. (2005). Static determination of allocation rates to support real-time garbage collection. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 193–202, New York, NY, USA. ACM.

[Meyer, 1992] Meyer, B. (1992). Applying "Design by Contract". *Computer*, 25(10):40–51.

[Müller et al., 2003] Müller, P., Poetzsch-Heffter, A., and Leavens, G. T. (2003). Modular Specification of Frame Properties in JML . *Concurrency and Computation: Practice and Experience*, 15:117–154.

[Mürk et al., 2007] Mürk, O., Larsson, D., and Hähnle, R. (2007). KeY-C: A tool for verification of C programs. In Pfenning, F., editor, *Proc. 21st Conference on Automated Deduction (CADE), Bremen, Germany*, volume 4603 of *LNCS*, pages 385–390. Springer-Verlag.

[Nilsen, 2006] Nilsen, K. (2006). A type system to assure scope safety within safety-critical Java modules. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 97–106, New York, NY, USA. ACM.

[Nilsen, 2008] Nilsen, K. (2008). Simple low-level real-time threading semantics to enable portability, efficiency, analyzability, and generality. In *JTRES '08: Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, pages 125–134, New York, NY, USA. ACM.

[Pham et al., 2008] Pham, T.-H., Truong, A.-H., Truong, N.-T., and Chin, W.-N. (2008). A fast algorithm to compute heap memory bounds of java card applets. In *SEFM '08: Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 259–267, Washington, DC, USA. IEEE Computer Society.

[Platzer, 2004] Platzer, A. (2004). An object-oriented dynamic logic with updates. Diplomarbeit, Universität Karlsruhe, Fakultät für Informatik. Institut für Logik, Komplexität und Deduktionssysteme. `http://i12www.ira.uka.de/~key/doc/2004/odlMasterThesis.pdf`.

[Platzer and Quesel, 2008] Platzer, A. and Quesel, J.-D. (2008). Keymaera: A hybrid theorem prover for hybrid systems (system description). In *IJCAR '08: Proceedings of the 4th international joint conference on Automated Reasoning*, pages 171–178, Berlin, Heidelberg. Springer-Verlag.

[Roth, 2006] Roth, A. (2006). *Specification and Verification of Object-Oriented Software Components*. PhD thesis, Universität Karlsruhe.

[Rothe, 2008] Rothe, M. (2008). Assisting the understanding of program behavior by using symbolic execution. Master's thesis, Chalmers University of Technology.

[Rountev et al., 2001] Rountev, A., Milanova, A., and Ryder, B. G. (2001). Points-to analysis for java using annotated constraints. *SIGPLAN Not.*, 36(11):43–55.

[RTCA, 1992] RTCA (1992). *Software considerations in airborne systems and equipment certification*. Washington DC. DO-178B.

[Rümmer, 2003] Rümmer, P. (2003). Ensuring the soundness of taclets – Constructing proof obligations for Java Card DL taclets. Studienarbeit, Fakultät für Informatik, Universität Karlsruhe.

[Rümmer, 2006] Rümmer, P. (2006). Sequential, parallel, and quantified updates of first-order structures. In *13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2006)*, LNCS 4246, pages 422–436. Springer.

[Schmitt and Weiß, 2007] Schmitt, P. H. and Weiß, B. (2007). Inferring invariants by symbolic execution. In Beckert, B., editor, *Proceedings, 4th International Verification Workshop (VERIFY'07)*, volume 259 of *CEUR Workshop Proceedings*, pages 195–210. CEUR-WS.org.

[Schoeberl et al., 2007] Schoeberl, M., Sondergaard, H., Thomsen, B., and Ravn, A. P. (2007). A profile for safety critical Java. In *ISORC '07: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 94–101, Washington, DC, USA. IEEE Computer Society.

[Siebert, 2007] Siebert, F. (2007). Realtime garbage collection in the JamaicaVM 3.0. In *JTRES '07: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pages 94–103, New York, NY, USA. ACM.

[Spoto and Poll, 2003] Spoto, F. and Poll, E. (2003). Static analysis for JML's assignable clauses. In Ghelli, G., editor, *Proceedings, 10th International Workshop on Foundations of Object-Oriented Languages (FOOL-10)*.

[Sălcianu and Rinard, 2005] Sălcianu, A. D. and Rinard, M. C. (2005). Purity and side effect analysis for Java programs. In Cousot, R., editor, *Proceedings, 6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2005)*, volume 3385 of *LNCS*, pages 199–215. Springer.

[Tofte and Talpin, 1997] Tofte, M. and Talpin, J.-P. (1997). Region-based memory management. *Information and Computation*.

[Ulbrich, 2007] Ulbrich, M. (2007). Software verification for Java 5. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe.

[Whaley and Rinard, 1999] Whaley, J. and Rinard, M. (1999). Compositional pointer and escape analysis for java programs. *SIGPLAN Not.*, 34(10):187–206.

[Zhao et al., 2004] Zhao, T., Noble, J., and Vitek, J. (2004). Scoped types for real-time Java. In *RTSS '04: Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 241–251, Washington, DC, USA. IEEE Computer Society.

Index