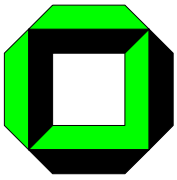
A Calculus for Data Abstraction

Roman Krenický

Diplomarbeit
Diploma Thesis



Universität Karlsruhe (TH)
Fakultät für Informatik
Institut für Theoretische Informatik

University of Karlsruhe (TH)
Department of Computer Science
Institute for Theoretical Computer Science

Betreuer: Dipl.-Inform. Mattias Ulbrich
Dipl.-Inform. Benjamin Weiß

Verantwortlicher Betreuer: Prof. Dr. Peter H. Schmitt

14. Juli 2009

Danksagung

An dieser Stelle möchte ich mich herzlich bei meinen Betreuern, Mattias Ulbrich und Benjamin Weiß, bedanken. Sie haben mich die ganze Diplomarbeit über hilfreich unterstützt, sowohl durch ihren fachlichen Rat und Betreuung als auch immer wieder durch erneute Motivation.

Vielen Dank insbesondere auch an meine Eltern. Sie haben mir dieses Studium überhaupt ermöglicht und mich immer unterstützt, ermuntert und mir Mut gemacht. Ohne sie wäre ich nicht so weit gekommen. Danke!

Authentizitätserklärung

Hiermit versichere ich, die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Roman Krenický
Karlsruhe, den 14. Juli 2009

Deutsche Zusammenfassung

Im Bereich der Programmverifikation stehen Informatiker immer wieder vor neuen Herausforderungen. Eine der wichtigsten ist als *frame problem* bekannt. Es ist die Frage danach, welche Teile des Programmzustands (der im wesentlichen im Heap gespeichert ist) durch eine gegebene Aktion verändert werden, oder vielmehr, welche *nicht*, bzw. wie diese Information abgeleitet oder spezifiziert werden kann. Im Zusammenhang mit abstrakten Spezifikationen, wie sie sich z. B. im objektorientierten Umfeld durch Vererbung ergeben können, bekommt das *frame problem* eine neue Qualität. Spezifikationen für Elternklassen sollen oft auch das Verhalten aller (potentiellen) Kindklassen umfassen, ohne letztere explizit zu kennen. Ein vielversprechender Lösungsansatz ist die Benutzung von Datenabstraktion zusammen mit expliziten Abhängigkeiten. Hierbei werden für Spezifikationszwecke spezielle Variablen, Funktionen oder Prädikate eingeführt, die keine konkreten Programmdaten modellieren (Datenabstraktion), und mithilfe spezieller Konstrukte kann man angeben, welche Daten oder Aktionen ihren Wert beeinflussen (explizite Abhängigkeiten). Dieser Ansatz wird u. a. für das Programmverifikationswerkzeug KeY in Erwägung gezogen.

Inhalt dieser Diplomarbeit ist die Untersuchung, inwiefern solche abstrakten Spezifikationsfunktionen (*Observer*) und expliziten Abhängigkeiten in KeY genutzt werden können, um auf syntaktischer Ebene logische Ausdrücke zu vereinfachen und mit rein syntaktischen Mitteln logische Schlüsse zu ziehen. Hierzu wird ein bereits eingesetztes Termersetzungssystem erweitert und anschließend analysiert.

Die Arbeit ist wie folgt strukturiert: Kapitel 1 führt kurz in die allgemeine Thematik ein. In Kapitel 2 werden die Grundlagen des KeY-Systems erläutert und der relevante Teil der zugrundeliegenden Logik formal definiert. Das Termersetzungssystem wird in Kapitel 3 wiedergegeben und seine Arbeitsweise in Kapitel 4 anhand einiger Beispiele präsentiert. In Kapitel 5 wird auf die Korrektheit des Termersetzungssystems eingegangen, und anschließend werden in Kapitel 6 seine Mächtigkeit sowie seine theoretischen Grenzen betrachtet. Kapitel 7 gibt einen Überblick über verwandte Literatur und Arbeiten, und Kapitel 8 schließt mit einer Zusammenfassung und einem Ausblick auf mögliche weiterführende Aufgaben.

Abstract

When trying to prove program correctness, verifiers have to struggle with different problems, a major one being the *frame problem*: knowing, which parts of the program state (mainly encoded in the heap) change by a given action, or rather which parts *do not*. The question becomes even harder, when abstraction is involved, for example, in an object-oriented context, by inheritance. Specifications for parent classes should be able to cover the behaviour of child classes, too, without any knowledge about the latter ones. A promising approach, which has come up in the recent years, is data abstraction in combination with explicit dependencies: specification-only variables, functions or predicates, which do not directly correspond to any program data, are introduced (data abstraction) and it is possible to specify, what other data or actions have an influence on their values (explicit dependencies). This approach is considered for the KeY program verification tool, using dynamic logic. This thesis examines, to what extent the abstract specification functions (*observers*) and explicit dependencies can be exploited to simplify logical expressions on a syntactical level. A term rewriting system is used for that task, its possibilities and theoretical limits are analysed.

Keywords: program verification, frame problem, modular verification, data abstraction, explicit dependencies, KeY tool, dynamic logic, updates

Contents

Chapter 1

Introduction

Since its early days, when the first serious attempts have been made to capture the meaning of programs formally and use the formalisation for correctness proofs [1, 2, 3], program verification has come a long way. The theory has continuously been examined and extended, new programming languages and paradigms have been approached and various verification tools have been developed. Yet, program verification is still an area of active research with different challenging problems (see, for instance, [4]).

One issue, which has come up quite early in artificial intelligence, is known as the *frame problem* [5]. The problem is, how to know and specify, what does not change by a specific action or piece of code, especially, if one does not want to introduce a vast number of *frame axioms* which specify for each bit of a program state, that it remains unchanged. In the context of modular programming or abstract programming concepts, for example inheritance and interfaces in object-oriented programming, the frame problem becomes even harder. In these settings, the program state is often not known as a whole – in modular programming due to encapsulation; with abstract concepts like inheritance and interfaces because there is an unlimited number of possible refinements, derivations or implementations. It is, however, desirable, to have a way of specifying and verifying things in these contexts, too. We want to be able to describe and check the behaviour of a module in a way which is independent of the code using it (as long as encapsulation is preserved, of course). We also want to be able to specify properties of interfaces and parent classes such that the behaviour of all implementing and derived classes is covered, too. In client code, it suffices to know the interface specification then, without having to bother which concrete implementation is used or maybe will be used in the future.

Various solutions have been proposed in the recent years, many of them based on data abstraction and explicit dependencies (e. g. [6], [7] or [8]). Data abstraction is involved in the sense that special variables, functions or predicates are introduced for the task of specification and verification only. That means, they do not correspond to any actual program data. Moreover, it is possible to specify what program data these specification variables depend on.

Think, for example, of a class that needs to store some kind of data. For some reason, we do not know or do not want to reveal what kind of data structure is used for that, but we want to be able to talk about it. To this end, we introduce

the specification-only variable *data*:

```
abstract class Memory
{
   spec var: data;

   abstract void store (element);   postcond: element in data
}
```

Now, a specific implementation might choose to use a list for storing the data:

```
class SomeMemory extends Memory
{
   List list;

   void store (element) {...}
}
```

In the context of `SomeMemory`, the specification variable *data* depends on the program variable `list` and all elements therein. We would specify this dependency explicitly in some fashion like:

data depends_on {`list`, *elements in* `list`}

One program verification tool which has to handle these problems is the KeY tool [9, 10], which is being developed at the University of Karlsruhe (Germany), the Chalmers University of Technology in Göteborg/Gothenburg (Sweden) and the University of Koblenz (Germany). It is used to verify programs written in Java. This is done by reasoning in a special kind of modal logic with actions, resulting from program statements, encoded in it. To address the issues of abstract and modular verification, it is intended to make special, dependent symbols and special predicates for expressing the dependencies part of the logic.

This thesis examines, in what way and to what extent these dependent symbols (called *observers*) and the explicit dependencies can be used to simplify expressions and reach verification results on the syntactic level. To this end, a term rewriting system is presented which transforms logical expressions like terms or formulae. Both, the possibilities and the theoretical limitations of this approach are analysed.

The rest of this thesis is structured as follows. In chapter 2, the concepts and the underlying logic used in KeY are explained. Based on that, the task of this thesis is formulated in a precise way. The rewriting system is presented in chapter 3. Chapter 4 demonstrates how the calculus works, giving some examples. In chapter 5, the correctness of the rewriting system is examined, while its strengths and limitations are considered in chapter 6. That chapter also describes a related utilisation of syntactic transformations to enable certain rules on the meta-level. Chapter 7 gives an overview of related work. Finally, in chapter 8, the work is summed up and a glance on possible future work is given.

Chapter 2

Setting

2.1 Basics

The KeY tool is used to verify Java programs and understands JML[1] [11] specifications. For verification, a dynamic logic [12] is used, which is a kind of modal logic with fragments of code embedded in the modalities $\Box$ and $\Diamond$ (hence, more precisely, dynamic logic is a *multi*-modal logic). The specific logic utilised in KeY is called JAVA CARD DL (or simply JAVA DL) and described in [9] in detail.

The basic idea is that a formula of the shape $[p]\varphi$ is true if and only if the formula φ is true after the execution of the program code p. In a deterministic context, $\langle p\rangle\varphi$ is almost the same, except that it includes the termination of p.[2] For example,

$$[\texttt{x++;}]x \doteq 2$$

is not universally valid, while the following formula is:

$$x \doteq 1 \;\rightarrow\; [\texttt{x++;}]x \doteq 2$$

Here, x is a program variable of integer type. Program variables are translated to nullary function symbols (i. e. constants) in terms, keeping logical and program variables apart (therefore, it is, for instance, not possible to quantify over program variables).

As usual for modal logics, the semantics is defined using a Kripke structure. There are several so-called *states*, each giving a first order interpretation of the underlying signature and a family of transition relations, specifying for each possible program (fragment) which states are reached from which ones by executing the program. The states of the Kripke structure represent program states (in the sense of the memory footprint). Executing the code fragment `x++;` in a state s_1 where the constant x has the value $x = 4$ will, for instance, result in a state s_2 which is equal to s_1 but with $x = 5$.

For reasoning about this kind of expression, programs inside of modalities are translated to so-called *updates* which are made part of the logic. Updates directly represent the changes of a program to the "memory" (i. e. to the logical

[1] Java Modeling Language.

[2] In general, $[p]\varphi$ means that φ is true after *all possible* executions of p – if there are several ones – while $\langle p\rangle\varphi$ means that *there is* an execution of p which makes φ true.

structure, resulting in transitions between states inside the Kripke structure). They are therefore easier to handle and independent of the programming language. In principle, updates have the shape of assignments. Examples of updates are "$a := a - 1$" or "$f(t) := c$". The statement `x++;` could, for instance, be translated to the update $x := x + 1$. The process of translation is done using symbolic execution. Applying an update in (or to) some state results in an *updated state*. Updates can be applied to formulae or terms (the syntax is $\{u\}\varphi$). The formula

$$\langle \texttt{x++;} \rangle x \doteq 2$$

could be translated to

$$\{x := x + 1\}(x \doteq 2)$$

Alternatively, one could say

$$(\{x := x + 1\}x) \doteq 2$$

as the constant 2 never changes.

When valuating the formula $x \doteq 1 \rightarrow \{x := x + 1\}\boxed{x} \doteq 2$ in a state s where x has the value 1, the framed x on the right hand side of the implication is valuated in the updated state with $x = 2$.

As the calculus presented in this thesis operates on the update level, programs and modalities are omitted in the following. Apart from that, typing is not considered (although handled by KeY) as it is not important for the presented work.

Let us turn to dependencies now. It turns out to be useful to have predicates and functions which depend on simple terms (or something like that). Consider, for example, the following auxiliary predicate on arrays:

$$\mathit{nonNullArray}(a) \;:\Leftrightarrow\; \forall i.a[i] \neq \mathit{null}$$

Arrays are modelled using an array access function. An array access like $a[i]$ is therefore actually something of the form $\mathit{array}(a, i)$. Clearly, for any array a the predicate $\mathit{nonNullArray}(a)$ depends on the array's entries (we do not worry about the length of the array in this simple example). Its value can only be changed indirectly by modifying the array. If the definition of this predicate was longer and more complicated, it would be handy to use the predicate instead of including the definition everywhere. The same thing can be done for function symbols. This leads to an important distinction resulting in two kinds of function symbols: *location function symbols* and *observer function symbols*. Location function symbols can be changed directly by occurring on the left hand side of an update (e. g. $f(a) := 22$). Observer function symbols depend on the values of location function symbols in certain points – e. g. the observer function symbol *obs* defined by $\mathit{obs} = f(0) + 2$, where f is a location function symbol, depends on $f(0)$. We call $f(0)$ a *location*. Observer function symbols cannot be updated directly. They can change along with the locations they depend on, only.

To be able to reason about such dependencies, we make them explicit. To this end, another kind of expression is introduced: *location descriptors*. Location descriptors allow to name sets of locations syntactically. An example of a location descriptor is $f(0)$, describing simply a location set with only one element: the location $f(0)$. Another example is *obs* (with *obs* being an observer

function symbol), which stands for all locations on which *obs* depends. Thus, in the example above, $f(0)$ and *obs* describe the same sets of locations (namely, the set $\{f(0)\}$). There are a few more constructors for location descriptors, allowing to express more complex location sets.

Moreover, there are special predicates, allowing to compare location descriptors. E. g. the fact, that the location descriptors $f(0)$ and *obs* name the same set of locations can be expressed in our logic explicitly as $\mathit{locEqual}(f(0), \mathit{obs})$.

In the following sections, the syntax and semantics of the logic in use will be specified formally. Apart from modalities and types, predicates are omitted, too. They can be modelled as truth-valued functions and would in fact be treated just like observer function symbols as far as the present work is concerned.

2.2 Syntax

Definition 2.1 (Signatures).
A *signature* Σ is a tuple

$$\Sigma = (\mathit{VSym}, \mathit{FSym}, \alpha)$$

where

- VSym is an infinite set of variable symbols
- FSym is a set of function symbols, consisting of two disjoint parts: $\mathit{FSym} = \mathit{FSym}_{loc} \cup \mathit{FSym}_{obs}$ with $\mathit{FSym}_{loc} \cap \mathit{FSym}_{obs} = \emptyset$ where
  - FSym_{loc} is the set of *location function symbols*
  - FSym_{obs} is the set of *observer function symbols*
- α assigns each function symbol an arity: $\alpha : \mathit{FSym} \to \mathbb{N}$

Let a signature Σ be given. Based on that, *expressions* (terms, formulae, location descriptors and updates) on Σ will be defined now. Notice, that for each kind an updated form (i. e. $\{u\}\alpha$) and explicit substitution applications ($\alpha[r/x]$) are defined.

Definition 2.2 (Terms).
Term_Σ, the set of terms, is the smallest set containing:

- x, for all $x \in \mathit{VSym}$
- $h(\bar{t})$, for all $h \in \mathit{FSym}$, $\bar{t} \in \mathit{Term}_\Sigma^{\alpha(h)}$
- if φ then t_1 else t_2, for all $\varphi \in \mathit{For}_\Sigma$, t_1,$t_2 \in \mathit{Term}_\Sigma$
- min $x.\varphi$, for all $x \in \mathit{VSym}$, $\varphi \in \mathit{For}_\Sigma$
- $\{u\}t$, for all $u \in \mathit{Upd}_\Sigma$, $t \in \mathit{Term}_\Sigma$
- $t[r/x]$, for all t,$r \in \mathit{Term}_\Sigma$, $x \in \mathit{VSym}$

Definition 2.3 (Formulae).
For_Σ, the set of formulae, is the smallest set containing:

- *true*
- *false*
- $\neg\varphi$, for all $\varphi \in For_\Sigma$
- $\varphi_1 \wedge \varphi_2$, for all φ_1,$\varphi_2 \in For_\Sigma$
- $\varphi_1 \vee \varphi_2$, for all φ_1,$\varphi_2 \in For_\Sigma$
- $\varphi_1 \rightarrow \varphi_2$, for all φ_1,$\varphi_2 \in For_\Sigma$
- $\forall x.\varphi$, for all $x \in VSym$, $\varphi \in For_\Sigma$
- $\exists x.\varphi$, for all $x \in VSym$, $\varphi \in For_\Sigma$
- $t_1 \doteq t_2$, for all t_1,$t_2 \in Term_\Sigma$
- $t_1 \dot{<} t_2$, for all t_1,$t_2 \in Term_\Sigma$
- $locSubset(ld_1, ld_2)$, for all ld_1,$ld_2 \in LocDesc_\Sigma$
- $locEqual(ld_1, ld_2)$, for all ld_1,$ld_2 \in LocDesc_\Sigma$
- $locDisjoint(ld_1, ld_2)$, for all ld_1,$ld_2 \in LocDesc_\Sigma$
- $\{u\}\varphi$, for all $u \in Upd_\Sigma$, $\varphi \in For_\Sigma$
- $\varphi[r/x]$, for all $\varphi \in For_\Sigma$, $x \in VSym$, $r \in Term_\Sigma$

Definition 2.4 (Location descriptors).
$LocDesc_\Sigma$, the set of location descriptors, is the smallest set containing:

- `empty`
- `everything`
- $h(\bar{t})$, for all $h \in FSym = FSym_{loc} \cup FSym_{obs}$, $\bar{t} \in Term_\Sigma^{\alpha(h)}$
- $ld_1 \| ld_2$, for all ld_1,$ld_2 \in LocDesc_\Sigma$
- `if` $\varphi.ld$, for all $\varphi \in For_\Sigma$, $ld \in LocDesc_\Sigma$
- `for` $x.ld$, for all $x \in VSym$, $ld \in LocDesc_\Sigma$
- $\{u\}ld$, for all $u \in Upd_\Sigma$, $ld \in LocDesc_\Sigma$
- $ld[r/x]$, for all $ld \in LocDesc_\Sigma$, $x \in VSym$, $r \in Term_\Sigma$

Definition 2.5 (Updates).
Upd_Σ, the set of updates, is the smallest set containing:

- `skip`
- $f(\bar{t}) := r$, for all $f \in FSym_{loc}$, $\bar{t} \in Term_\Sigma^{\alpha(f)}$, $r \in Term_\Sigma$
- $u_1 \| u_2$, for all u_1,$u_2 \in Upd_\Sigma$
- $u_1; u_2$, for all u_1,$u_2 \in Upd_\Sigma$

- `if` $\varphi.u$, for all $\varphi \in For_\Sigma$, $u \in Upd_\Sigma$
- `for` $x.u$, for all $x \in VSym$, $u \in Upd_\Sigma$
- $ld := *n$, for all $ld \in LocDesc_\Sigma$, $n \in \mathbb{N}$
- $\{u'\}u$, for all u,$u' \in Upd_\Sigma$
- $u[r/x]$, for all $u \in Upd_\Sigma$, $x \in VSym$, $r \in Term_\Sigma$

Let us have a brief look at the different constructors for location descriptors and updates – apart from the base cases ($h(\bar{t})$ and $f(\bar{t}) := r$). Two updates can be combined using parallel composition ($u_1 \| u_2$), resulting in an update which unites the effects of the two original updates. Conditional updates (`if` $\varphi.u$) are applied only if the condition is met. Moreover, updates can be quantified (e. g. `for` $x.(f(x) := x^2)$). Analogously, there are parallel, conditional and quantified location descriptors. There are, furthermore, the special location descriptors `empty` and `everything` standing for the empty location set and the set of all possible locations, respectively. There is also the empty update `skip` which does not update anything.

There are two different ways to combine two updates: parallel and sequential composition. With sequential composition ($u_1; u_2$), first update u_1 is applied and then – in the updated state – update u_2. That means, that u_2 is valuated after the effects of u_1 have taken place. In contrast to that, with parallel composition ($u_1 \| u_2$), both updates are applied at the same time, i. e. u_1 does not affect the valuation of u_2. Consider, for example, two constants $x = 2$ and $y = 5$ and the updates $x := y$ and $y := x$ (figure 2.1). Applying $x := y; y := x$ will result in $x = 5$ and $y = 5$ because at the time, when the second update ($y := x$) is applied, the value of x is already 5. On the other hand, the application of $x := y \| y := x$ yields $x = 5$ and $y = 2$, i. e. the values of x and y are exchanged.

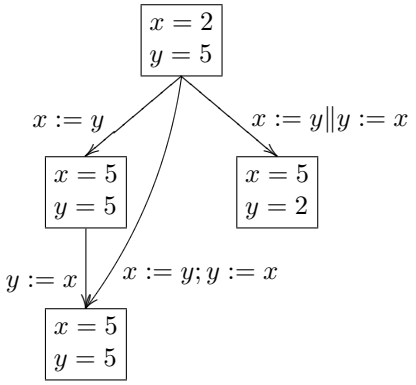


Figure 2.1: Comparison of sequential and parallel composition of updates.

Finally, there are *anonymising updates* of the form $ld := *n$ (e. g. $f(c) := *3$). They are used to express arbitrary updates on specific locations. The formula $\{(\texttt{for}\ x.f(x)) := *3\}\varphi$, for instance, expresses in any state that φ is true, no matter how the values of $f(x)$ for any x are changed. The number is used to identify different anonymising updates: $\big(\{l := *3\}l\big) \doteq \big(\{l := *3\}l\big)$ is universally valid, while $\big(\{l := *3\}l\big) \doteq \big(\{l := *4\}l\big)$ is not.

2.3 Semantics

Definition 2.6 (Kripke structures).
A *Kripke structure* $\mathcal{K}$ for a given signature Σ is a tuple

$$\mathcal{K} = (\mathcal{D}, \prec, \mathcal{S}, \mathit{observe}, \mathit{depends}, *)$$

where

- $\mathcal{D}$ is a non-empty set (the *domain* or *universe* of $\mathcal{K}$)
- $\prec$ is a strict well-ordering on $\mathcal{D}$, i. e. $\prec \subseteq \mathcal{D}^2$ s. t. $\prec$ is transitive, irreflexive and every non-empty subset of $\mathcal{D}$ has a least element w. r. t. $\prec$ (it follows that $\prec$ obeys the law of trichotomy)
- $\mathcal{S}$ is the set of *all possible* interpretations of FSym_{loc} of Σ where each location function symbol is interpreted as a function in $\mathcal{D}$:

  for all $s \in \mathcal{S}$ and $f \in \mathit{FSym}_{loc}$:

  $$s(f) : \mathcal{D}^{\alpha(f)} \to \mathcal{D}$$

  elements of $\mathcal{S}$ are called *states* or *worlds*
- *depends* defines, for each state, the dependencies of each observer function symbol of Σ:

  for all $s \in \mathcal{S}$ and $g \in \mathit{FSym}_{obs}$:[3]

  $$\mathit{depends}(s, g) : \mathcal{D}^{\alpha(g)} \to 2^{\mathit{Locations}_{\mathcal{K}}}$$

- *observe* gives, for each state, an interpretation for each observer function symbol of Σ:

  for all $s \in \mathcal{S}$ and $g \in \mathit{FSym}_{obs}$:

  $$\mathit{observe}(s, g) : \mathcal{D}^{\alpha(g)} \to \mathcal{D}$$

  where *observe* must respect the observer's dependencies in the following way:

  for all s,$s' \in \mathcal{S}$, $g \in \mathit{FSym}_{obs}$ and $\bar{d} \in \mathcal{D}^{\alpha(g)}$ let

  $$L := \mathit{depends}(s, g)(\bar{d})$$
  $$L' := \mathit{depends}(s', g)(\bar{d})$$

  now:[4]

  $$s \approx_L s' \quad \text{or} \quad s \approx_{L'} s'$$
  $$\Rightarrow$$
  $$\mathit{observe}(s, g)(\bar{d}) = \mathit{observe}(s', g)(\bar{d})$$

- $*$ is a function: $* : \mathbb{N} \to \mathcal{S}$

[3] For $\mathit{Locations}_{\mathcal{K}}$ see definition 2.8.
[4] For $\approx_L$ see definition 2.9.

We require an explicit ordering on the domain $\mathcal{D}$ of a Kripke structure. The reason is that we need it for the semantics of updates. An update like $f(t) := c$ means, intuitively, that the value of c is assigned to $f(t)$. Or, $x := y \| y := x$ assigns the value of x to y and vice versa at the same time – i. e. the values of x and y are swapped. But what is the meaning of $x := 1 \| x := 5$? There is a conflict (or *clash*) in that update, as, clearly, x cannot be assigned the values 1 and 5 at the same time.

These cases are handled by introducing an implicit clash resolution in the semantics. For parallel updates a *last win semantics* is used: later assignments override earlier ones. In the example of $x := 1 \| x := 5$ that means that the second assignment $x := 5$ wins and x is assigned the value 5. Note, that this still does not give the semantics of sequential updates. Both parts of a parallel update are still valuated at the same time and only then clashes are resolved. That means that the result of $x := x + 1 \| x := x + 2$ is an incrementation of x by 2, as the update $x := x + 1$ is overridden by $x := x + 2$. On the other hand, $x := x + 1; x := x + 2$ (i. e. sequential composition) first increments x by 1 and subsequently additionally increments it by 2. All in all, the result is an incrementation by 3.

Clashes can occur in quantified updates as well. Consider a function f on natural numbers and the update $\texttt{for}\ n.(f(n/3) := n))$ where / is the division on integers. The update yields the following assignments: $f(0) := 0$, $f(0) := 1$, $f(0) := 2$, $f(1) := 3$, $f(1) := 4$, $f(1) := 5$, $f(2) := 6$, ... As the assignments are 'generated' by the quantification, we need an ordering on the domain to be able to talk about "earlier" and "later" assignments. The semantics used in the case of quantified updates is the *least witness wins semantics*: for clashes, the assignment generated by the least value for the quantification variable (n in our example) wins. Choosing the usual order relation $<$ on numbers (which indeed is a well-ordering on $\mathbb{N}$), that means that, for example, the assignment $f(0) := 0$ (for $n = 0$) overrides $f(0) := 1$ (for $n = 1$) and $f(0) := 2$ (for $n = 2$). Hence, the quantified update results in the assignments: $f(0) := 0$, $f(1) := 3$, $f(2) := 6$, ...

Let us continue with the formal definitions. For all of the following definitions, let a signature $\Sigma = (\mathit{VSym}, \mathit{FSym}, \alpha)$ (with $\mathit{FSym} = \mathit{FSym}_{loc} \cup \mathit{FSym}_{obs}$) and a Kripke structure $\mathcal{K} = (\mathcal{D}, \prec, \mathcal{S}, \mathit{observe}, \mathit{depends}, *)$ for Σ be given.

Definition 2.7 (Locations).
A *location* is any tuple $(f, \bar{d})$ with $f \in \mathit{FSym}_{loc}$ and $\bar{d} \in \mathcal{D}^{\alpha(f)}$.

As a shorthand notation we can write $s(l)$ instead of $s(f)(\bar{d})$ for $l = (f, \bar{d})$, $s \in \mathcal{S}$ which gives the value of location l in the state s.

Definition 2.8 ($\mathit{Locations}_{\mathcal{K}}$).
We define the set of all locations for $\mathcal{K}$ as follows:

$$\mathit{Locations}_{\mathcal{K}} := \{(f, \bar{d}) \mid f \in \mathit{FSym}_{loc}, \bar{d} \in \mathcal{D}^{\alpha(f)}\}$$

Definition 2.9 (Equivalence w. r. t. location sets).
We define the equivalence of states $\approx_L \subseteq \mathcal{S}^2$ with respect to some set L of locations ($L \subseteq \mathit{Locations}_{\mathcal{K}}$).
For all s,$s' \in \mathcal{S}$:

$$s \approx_L s' \ :\Leftrightarrow\ s(l) = s'(l) \text{ for all } l \in L$$

Clearly, $\approx_L$ is an equivalence relation for any L.

Definition 2.10 (Semantic updates).
A *semantic update* U is any (possibly empty) set of tuples (l, d) where $l \in \mathit{Locations}_\mathcal{K}$ and $d \in \mathcal{D}$ and where for any $(l_1, d_1), (l_2, d_2) \in L$ the implication $l_1 = l_2 \Rightarrow d_1 = d_2$ holds (i. e. there are no "conflicting" pairs, semantic updates are *consistent*).

We define the application of semantic updates to states ($U : \mathcal{S} \to \mathcal{S}$) as follows:

$$U(s)(l) := \begin{cases} d & , \ (l,d) \in U \\ s(l) & , \ \text{otherwise} \end{cases} \qquad \text{for all } l \in \mathit{Locations}_\mathcal{K}$$

$\mathit{Updates}_\mathcal{K}$ denotes the set of all semantic updates for $\mathcal{K}$.

Definition 2.11 (Variable assignments).
Variable assignments β are defined as usual:

$$\beta : \mathit{VSym} \to \mathcal{D}$$

For a variable assignment β, a fixed $x \in \mathit{VSym}$ and $d \in \mathcal{D}$ the modified variable assignment β_x^d is defined as:

$$\beta_x^d(z) := \begin{cases} d & , \ z = x \\ \beta(z) & , \ \text{otherwise} \end{cases} \qquad \text{for all } z \in \mathit{VSym}$$

Definition 2.12 ($\min_\prec$).
Let $\min_\prec$ be the minimum operator on $2^\mathcal{D} \setminus \emptyset$ w. r. t. $\prec$, i. e. for any $X \subseteq \mathcal{D}$, $X \neq \emptyset$

$\min_\prec X = m \in X$

with

$m \prec d$ for all $d \in X \setminus \{m\}$.

Such an m exists for all X, as $\prec$ is a well-ordering on $\mathcal{D}$.

The following definitions give the semantics of expressions by defining their valuations for a given state s of $\mathcal{K}$ and a variable assignment β for $\mathcal{K}$ and Σ.

Definition 2.13 (Semantics of terms).
Terms are valuated by the term valuation function $\mathit{termVal}_{\mathcal{K},s,\beta} : \mathit{Term}_\Sigma \to \mathcal{D}$ which is defined as follows:

- $\mathit{termVal}_{\mathcal{K},s,\beta}(x) := \beta(x), \quad x \in \mathit{VSym}$
- $\mathit{termVal}_{\mathcal{K},s,\beta}(h(\bar{t})) := \begin{cases} s(h)(\mathit{termVal}_{\mathcal{K},s,\beta}(\bar{t})) & , \ h \in \mathit{FSym}_{loc} \\ \mathit{observe}(s,h)(\mathit{termVal}_{\mathcal{K},s,\beta}(\bar{t})) & , \ h \in \mathit{FSym}_{obs} \end{cases}$
- $\mathit{termVal}_{\mathcal{K},s,\beta}(\text{if } \varphi \text{ then } t_1 \text{ else } t_2) := \begin{cases} \mathit{termVal}_{\mathcal{K},s,\beta}(t_1) & , \ (\mathcal{K},s,\beta) \models \varphi \\ \mathit{termVal}_{\mathcal{K},s,\beta}(t_2) & , \ (\mathcal{K},s,\beta) \not\models \varphi \end{cases}$
- $\mathit{termVal}_{\mathcal{K},s,\beta}(\min\, x.\varphi) := \begin{cases} \min_\prec \underbrace{\{d \in \mathcal{D} \mid (\mathcal{K},s,\beta_x^d) \models \varphi\}}_{=:\, M} & , \ M \neq \emptyset \\ \min_\prec \mathcal{D} & , \ M = \emptyset \end{cases}$
- $\mathit{termVal}_{\mathcal{K},s,\beta}(\{u\}t) := \mathit{termVal}_{\mathcal{K},s',\beta}(t)$
  where $s' = \mathit{updVal}_{\mathcal{K},s,\beta}(u)(s)$

- $termVal_{\mathcal{K},s,\beta}(t[r/x]) := termVal_{\mathcal{K},s,\beta'}(t)$
  where $\beta' = \beta_x^{termVal_{\mathcal{K},s,\beta}(r)}$

In expressions like $termVal_{\mathcal{K},s,\beta}(\bar{t})$ with $\bar{t} \in Term_\Sigma{}^*$ the application of $termVal_{\mathcal{K},s,\beta}$ is to be understood element-wise, i. e. $termVal_{\mathcal{K},s,\beta}((t_1, t_2, \dots)) = (termVal_{\mathcal{K},s,\beta}(t_1), termVal_{\mathcal{K},s,\beta}(t_2), \dots)$.

Definition 2.14 (Semantics of formulae).
Formulae are valuated by the semantical truth relation $\models \subseteq \{(\mathcal{K}, s, \beta)\} \times For_\Sigma$ which is defined as follows:

- $(\mathcal{K}, s, \beta) \models true$
- $(\mathcal{K}, s, \beta) \not\models false$
- $(\mathcal{K}, s, \beta) \models \neg\varphi \;:\Leftrightarrow\; (\mathcal{K}, s, \beta) \not\models \varphi$
- $(\mathcal{K}, s, \beta) \models \varphi_1 \wedge \varphi_2 \;:\Leftrightarrow\; (\mathcal{K}, s, \beta) \models \varphi_1$ and $(\mathcal{K}, s, \beta) \models \varphi_2$
- $(\mathcal{K}, s, \beta) \models \varphi_1 \vee \varphi_2 \;:\Leftrightarrow\; (\mathcal{K}, s, \beta) \models \varphi_1$ or $(\mathcal{K}, s, \beta) \models \varphi_2$
- $(\mathcal{K}, s, \beta) \models \varphi_1 \rightarrow \varphi_2 \;:\Leftrightarrow\; (\mathcal{K}, s, \beta) \not\models \varphi_1$ or $(\mathcal{K}, s, \beta) \models \varphi_2$
- $(\mathcal{K}, s, \beta) \models \forall x.\varphi \;:\Leftrightarrow\; (\mathcal{K}, s, \beta_x^d) \models \varphi$ for all $d \in \mathcal{D}$
- $(\mathcal{K}, s, \beta) \models \exists x.\varphi \;:\Leftrightarrow\; (\mathcal{K}, s, \beta_x^d) \models \varphi$ for some $d \in \mathcal{D}$
- $(\mathcal{K}, s, \beta) \models t_1 \doteq t_2 \;:\Leftrightarrow\; termVal_{\mathcal{K},s,\beta}(t_1) = termVal_{\mathcal{K},s,\beta}(t_1)$
- $(\mathcal{K}, s, \beta) \models t_1 \dot{\prec} t_2 \;:\Leftrightarrow\; termVal_{\mathcal{K},s,\beta}(t_1) \prec termVal_{\mathcal{K},s,\beta}(t_1)$
- $(\mathcal{K}, s, \beta) \models locSubset(ld_1, ld_2) \;:\Leftrightarrow\; locVal_{\mathcal{K},s,\beta}(ld_1) \subseteq locVal_{\mathcal{K},s,\beta}(ld_2)$
- $(\mathcal{K}, s, \beta) \models locEqual(ld_1, ld_2) \;:\Leftrightarrow\; locVal_{\mathcal{K},s,\beta}(ld_1) = locVal_{\mathcal{K},s,\beta}(ld_2)$
- $(\mathcal{K}, s, \beta) \models locDisjoint(ld_1, ld_2) \;:\Leftrightarrow\; locVal_{\mathcal{K},s,\beta}(ld_1) \cap locVal_{\mathcal{K},s,\beta}(ld_2) = \emptyset$
- $(\mathcal{K}, s, \beta) \models \{u\}\varphi \;:\Leftrightarrow\; (\mathcal{K}, s', \beta) \models \varphi$
  where $s' = updVal_{\mathcal{K},s,\beta}(u)(s)$
- $(\mathcal{K}, s, \beta) \models \varphi[r/x] \;:\Leftrightarrow\; (\mathcal{K}, s, \beta') \models \varphi$
  where $\beta' = \beta_x^{termVal_{\mathcal{K},s,\beta}(r)}$

Alternatively, the relation $\models$ with the left hand side fixed could be regarded as a unary valuation function, too: $(\mathcal{K}, s, \beta) \models \cdot : For_\Sigma \rightarrow \{\top, \bot\}$.

Definition 2.15 (Semantics of location descriptors).
Location descriptors are valuated by the location descriptor valuation function $locVal_{\mathcal{K},s,\beta} : LocDesc_\Sigma \rightarrow 2^{Locations_\mathcal{K}}$ which is defined as follows:

- $locVal_{\mathcal{K},s,\beta}(\texttt{empty}) := \emptyset$
- $locVal_{\mathcal{K},s,\beta}(\texttt{everything}) := Locations_\mathcal{K}$
- $locVal_{\mathcal{K},s,\beta}(h(\bar{t})) := \begin{cases} \{\ (h, termVal_{\mathcal{K},s,\beta}(\bar{t}))\ \} & ,\ h \in FSym_{loc} \\ depends(s,h)(termVal_{\mathcal{K},s,\beta}(\bar{t})) & ,\ h \in FSym_{obs} \end{cases}$

- $locVal_{\mathcal{K},s,\beta}(ld_1 \| ld_2) := locVal_{\mathcal{K},s,\beta}(ld_1) \cup locVal_{\mathcal{K},s,\beta}(ld_2)$
- $locVal_{\mathcal{K},s,\beta}(\texttt{if } \varphi.ld) := \begin{cases} locVal_{\mathcal{K},s,\beta}(ld) & , (\mathcal{K},s,\beta) \models \varphi \\ \emptyset & , (\mathcal{K},s,\beta) \not\models \varphi \end{cases}$
- $locVal_{\mathcal{K},s,\beta}(\texttt{for } x.ld) := \bigcup_{d \in \mathcal{D}} locVal_{\mathcal{K},s,\beta^d_x}(ld)$
- $locVal_{\mathcal{K},s,\beta}(\{u\}ld) := locVal_{\mathcal{K},s',\beta}(ld)$
  where $s' = updVal_{\mathcal{K},s,\beta}(u)(s)$
- $locVal_{\mathcal{K},s,\beta}(ld[r/x]) := locVal_{\mathcal{K},s,\beta'}(ld)$
  where $\beta' = \beta_x^{termVal_{\mathcal{K},s,\beta}(r)}$

Definition 2.16 (Semantics of updates).
Updates are valuated by the update valuation function $updVal_{\mathcal{K},s,\beta} : Upd_\Sigma \to Updates_\mathcal{K}$ which is defined as follows:

- $updVal_{\mathcal{K},s,\beta}(\texttt{skip}) := \emptyset$
- $updVal_{\mathcal{K},s,\beta}(f(\bar{t}) := r) := \{\ (\,(f, termVal_{\mathcal{K},s,\beta}(\bar{t})),\ termVal_{\mathcal{K},s,\beta}(r)\,)\ \}$
- $updVal_{\mathcal{K},s,\beta}(u_1 \| u_2) := (\underbrace{updVal_{\mathcal{K},s,\beta}(u_1)}_{=:\,U_1} \cup \underbrace{updVal_{\mathcal{K},s,\beta}(u_2)}_{=:\,U_2}) \setminus C$
  where $C = \{(l,d) \in U_1 \mid (l,d') \in U_2 \text{ for some } d' \neq d\}$
- $updVal_{\mathcal{K},s,\beta}(u_1; u_2) := (\underbrace{updVal_{\mathcal{K},s,\beta}(u_1)}_{=:\,U_1} \cup \underbrace{updVal_{\mathcal{K},s',\beta}(u_2)}_{=:\,U_2'}) \setminus C'$
  where $C' = \{(l,d) \in U_1 \mid (l,d') \in U_2' \text{ for some } d' \neq d\}$
  $s' = updVal_{\mathcal{K},s,\beta}(u_1)(s)$
- $updVal_{\mathcal{K},s,\beta}(\texttt{if } \varphi.u) := \begin{cases} updVal_{\mathcal{K},s,\beta}(u) & , (\mathcal{K},s,\beta) \models \varphi \\ \emptyset & , (\mathcal{K},s,\beta) \not\models \varphi \end{cases}$
- $updVal_{\mathcal{K},s,\beta}(\texttt{for } x.u) := \{\ (l,a) \in updVal_{\mathcal{K},s,\beta^d_x}(u) \mid d \in \mathcal{D} \wedge [\,(l,a') \in updVal_{\mathcal{K},s,\beta^{d'}_x}(u) \wedge a \neq a' \to d \prec d'\,]\ \}$
- $updVal_{\mathcal{K},s,\beta}(ld := *n) := \{(l, *(n)(l)) \mid l \in locVal_{\mathcal{K},s,\beta}(ld)\}$
- $updVal_{\mathcal{K},s,\beta}(\{u'\}u) := updVal_{\mathcal{K},s',\beta}(u)$
  where $s' = updVal_{\mathcal{K},s,\beta}(u')(s)$
- $updVal_{\mathcal{K},s,\beta}(u[r/x]) := updVal_{\mathcal{K},s,\beta'}(u)$
  where $\beta' = \beta_x^{termVal_{\mathcal{K},s,\beta}(r)}$

Note the clash resolution for parallel, sequential and quantified updates, which ensures that the resulting semantic updates are consistent.

Definition 2.17 (Equivalence of expressions).
Let a signature Σ be given. Two expressions of Σ are equivalent $\equiv$, if they always have the same semantics:

$$\begin{aligned} \alpha \equiv \alpha' \quad :\Leftrightarrow \quad & val_{\mathcal{K},s,\beta}(\alpha) = val_{\mathcal{K},s,\beta}(\alpha') \\ & \text{for all } \mathcal{K} \text{ for } \Sigma, s \in \mathcal{S} \text{ of } \mathcal{K} \\ & \text{and } \beta \text{ for } \mathcal{K} \text{ and } \Sigma \end{aligned}$$

where α,$\alpha' \in Term_\Sigma$ or For_Σ or $LocDesc_\Sigma$ or Upd_Σ and $val_{\mathcal{K},s,\beta}()$ is the valuation on terms ($termVal_{\mathcal{K},s,\beta}()$), formulae ($(\mathcal{K},s,\beta) \models \cdot$), location descriptors ($locVal_{\mathcal{K},s,\beta}()$) or updates ($updVal_{\mathcal{K},s,\beta}()$), respectively.

Proposition 2.1. *$\equiv$ is a congruence relation on $Term_\Sigma \cup For_\Sigma \cup LocDesc_\Sigma \cup Upd_\Sigma$ with respect to all constructors.*

2.4 Update Simplifier

When reasoning in our dynamic logic with updates, the updates add complexity compared to the well-explored reasoning in first order logic. It is therefore quite natural to try and get rid of updates in expressions as soon as possible. To this end, Philipp Rümmer developed in [13] a term rewriting system, the *update simplifier*, which simplifies and applies updates on the syntactic level. The update simplifier consists of a set of rules of the form "$lhs \rightsquigarrow rhs$" where both sides are expressions of the same kind (i. e. terms, formulae, location descriptors or updates) and are equivalent ($lhs \equiv rhs$). Examples of such rules are:

$$\{u\}f(\bar{t}) \quad \rightsquigarrow \quad (\{u\}f)(\{u\}\bar{t}) \tag{2.1}$$

$$(\{f(\bar{s}) := r\}f)(\bar{t}) \quad \rightsquigarrow \quad \text{if } \bar{t} \doteq \bar{s} \text{ then } r \text{ else } f(\bar{t}) \tag{2.2}$$

$$(\{f'(\bar{s}) := r\}f)(\bar{t}) \quad \rightsquigarrow \quad f(\bar{t}) \quad , f \neq f' \tag{2.3}$$

for location function symbols f, f'. The update simplifier works *locally*, which means that it does not consider the context of expressions. The rewritten expressions (the left hand sides of the rules) could be sub-expressions of larger expressions or various axioms could be present. All these things are not considered by the update simplifier, its rules are applicable as soon as an expression matches the left hand side of a rule. This is sound, because $\equiv$ is a congruence relation (see proposition 2.1). It is also the locality and the restriction to the syntactic level, which make the term rewriting approach fast and thus attractive.

Here is a small example of how the update simplifier works. Numbers (and in particular 0) are nullary function symbols and the framed expressions are the ones to which the next rule is applied.

$$\Big(\boxed{\{f(0) := 4\}\ f(0)}\Big) \ \doteq\ 7$$

$$\overset{(2.1)}{\rightsquigarrow}$$

$$\Big(\ (\{f(0) := 4\}\ f)(\boxed{\{f(0) := 4\}0})\ \Big) \ \doteq\ 7$$

$$\overset{(2.3)}{\rightsquigarrow}$$

$$\Big(\boxed{(\{f(0) := 4\}\ f)(0)}\Big) \ \doteq\ 7$$

$$\overset{(2.2)}{\rightsquigarrow}$$

$$\big(\text{if } (0 \doteq 0) \text{ then } 4 \text{ else } f(0)\big) \ \doteq\ 7$$

Now two simple steps (not performed by the update simplifier but in the course of generic term simplification) yield "$4 \doteq 7$" which again can easily be transformed to *false*.

An important fact is, that the update simplifier successfully reaches the desired goal: updates are always eliminated entirely, i. e. expressions containing updates are rewritten to equivalent ones without updates. The result of this "update simplification" are first order logic expressions. (It should be noted, that the update simplifier can also be invoked while formulae contain modalities – box or diamond – in which case updates may remain until the modalities are not removed by symbolic execution.)

At the time of development, however, the logic in use did not contain observer function symbols, explicit dependencies, location descriptors and anonymising updates. It is the topic of this thesis to extend the update simplifier to handle these new concepts.

2.5 Abstract Specifications

We have already seen one example of where observers can be useful: the *nonNullArray* predicate in section 2.1.

Another, more important case are abstract specifications, that means specifications, which will be refined for real usage and may be refined in different, customised ways. Thinking of object-oriented languages like Java, that might involve interfaces and abstract methods and possible contracts for them. Although these specifications are not complete in some (intended) way, they may have properties which we want to and should be able to prove. While observers and explicit dependencies were a convenience for auxiliary predicates and functions like *nonNullArray*, they are crucial for reasoning about abstract specifications.

Let us regard the example of a Java interface:

```
interface IntStack
{
   public void push (int x);
   public int top ();
   ...
}
```

This is intended to be the interface for a stack of integers. The method `top()` should always return the same value as long as the stack is not modified. We expect `push()` to modify the stack, on the other hand. Now, a possible imple-

mentation of this interface would be:

```
class ArrIntStack implements IntStack
{
   int intArray[];

   public void push (int x)
   { ... }
   public int top ()
   { ... }
   ...
}
```

We could specify the desired properties for every implementation and use them in proofs for other code that uses such a stack. We would have to make a new proof for every implementation used. But that is not what we want. We want to be able to specify the properties at an abstract level, that means for the interface. Then, we could use them in proofs of code that uses some arbitrary implementation of the interface without knowing what that implementation looks like. The problem in the case of our example is, of course, that we want to talk about the contents of the stack, but we do not know anything about the underlying data store. It will be defined by the implementations, only.

JML provides a solution to this problem: *model fields*. We can specify an abstract data store for the use of the specification only. This allows us to specify the desired properties:

```
interface IntStack
{
   //@ public model JMLValueSequence datastore;

   //@ modifies datastore;
   public void push (int x);

   //@ accessible datastore;
   //@ modifies \nothing;
   public int top ();
   ...
}
```

The `accessible` clause for `top()` specifies that the method's result depends on `datastore` only – as long as `datastore` does not change, the result of `top()` remains the same. Now, this specification can be used in proofs for code that uses an implementation of `IntStack`.

For a specific implementation (e. g. `ArrIntStack`), we need to link the im-

plementation (`intArray[]`) with the abstraction (`datastore`):

```
class ArrIntStack implements IntStack
{
   int intArray[]; //@ in datastore;
                   //@ maps intArray[*] into datastore;

   public void push (int x)
   { ...; intArray[intArray.length-1] = x; }

   public int top ()
   { return intArray[intArray.length-1]; }
   ...
}
```

This mechanism is translated to our logic as follows: the concrete data locations (the attribute `intArray` and its entries) are modelled as location function symbols. The model fields (`datastore` in this example) are modelled as observer function symbols.

What we need, yet, is a way of specifying dependencies explicitly. This is done using the predicates *locEqual*, *locSubset* and *locDisjoint* and stating appropriate axioms. To say, for example, that an observer g depends on the locations l and $f(c)$ we would state $\mathit{locSubset}(l \| f(c), g)$. If we wanted g to depend on these locations *only*, we would use *locEqual* instead. We can also express the opposite: $\mathit{locDisjoint}(g, f(c))$ says that g does *not* depend on $f(c)$.

For the example above, the connection between `datastore` and `intArray` would be established for all objects o of type `ArrIntStack` by

$$\forall\, o\,.\, \mathit{locEqual}(o.\mathit{datastore}, o.\mathit{intArray} \| \texttt{for}\ i\,.\,(o.\mathit{intArray}[i]))$$

This would allow to check, whether the implementation of `ArrIntStack` complies with the specification of `IntStack`.

On the semantic level, the axiom restricts the set of considered Kripke structures to the ones with appropriate *depends*() functions.

2.6 New Expressions

For technical purposes, further expressions have been added. First of all, non-recursive update application to function symbols $(\{u\}h)(\bar{t})$, where only the function symbol (h) is valuated in the updated state, while the arguments $(\bar{t})$ are valuated in the original state.[5] Then, a function giving the domain of an update as a location descriptor has been added $(\mathit{dom}(u))$. Furthermore, intersection $(\cap)$ and difference $(\setminus)$ of location descriptors have been defined. Finally, the RESTRICT constructor has been introduced, which restricts the domain of an update $(\mathrm{RESTRICT}(u, ld))$. In this section, the definition of the syntax and the semantics will be extended to include these expressions as well.

[5] For terms this is exactly the NON-REC constructor in [13].

2.6.1 Syntax

$Term_\Sigma$ additionally contains:

- $(\{u\}h)(\bar{t})$, for all $u \in Upd_\Sigma$, $h \in FSym$, $\bar{t} \in Term_\Sigma^{\alpha(h)}$

$LocDesc_\Sigma$ additionally contains:

- $(\{u\}h)(\bar{t})$, for all $u \in Upd_\Sigma$, $h \in FSym$, $\bar{t} \in Term_\Sigma^{\alpha(h)}$
- $ld_1 \cap ld_2$, for all $ld_1, ld_2 \in LocDesc_\Sigma$
- $ld_1 \setminus ld_2$, for all $ld_1, ld_2 \in LocDesc_\Sigma$
- $dom(u)$, for all $u \in Upd_\Sigma$

Upd_Σ additionally contains:

- RESTRICT(u, ld), for all $u \in Upd_\Sigma$, $ld \in LocDesc_\Sigma$

2.6.2 Semantics

Terms:

- $termVal_{\mathcal{K},s,\beta}((\{u\}h)(\bar{t})) := \begin{cases} s'(h)(termVal_{\mathcal{K},s,\beta}(\bar{t})) & , \ h \in FSym_{loc} \\ observe(s', h)(termVal_{\mathcal{K},s,\beta}(\bar{t})) & , \ h \in FSym_{obs} \end{cases}$
  where $s' = updVal_{\mathcal{K},s,\beta}(u)(s)$

Location descriptors:

- $locVal_{\mathcal{K},s,\beta}((\{u\}h)(\bar{t})) := \begin{cases} locVal_{\mathcal{K},s,\beta}(h(\bar{t})) & , \ h \in FSym_{loc} \\ depends(s', h)(termVal_{\mathcal{K},s,\beta}(\bar{t})) & , \ h \in FSym_{obs} \end{cases}$
  where $s' = updVal_{\mathcal{K},s,\beta}(u)(s)$
- $locVal_{\mathcal{K},s,\beta}(ld_1 \cap ld_2) := locVal_{\mathcal{K},s,\beta}(ld_1) \cap locVal_{\mathcal{K},s,\beta}(ld_2)$
- $locVal_{\mathcal{K},s,\beta}(ld_1 \setminus ld_2) := locVal_{\mathcal{K},s,\beta}(ld_1) \setminus locVal_{\mathcal{K},s,\beta}(ld_2)$
- $locVal_{\mathcal{K},s,\beta}(dom(u)) := \{l \mid (l, d) \in updVal_{\mathcal{K},s,\beta}(u) \text{ for some } d \in \mathcal{D}\}$

Updates:

- $updVal_{\mathcal{K},s,\beta}(\text{RESTRICT}(u, ld))$
  $:= \{(l, d) \in updVal_{\mathcal{K},s,\beta}(u) \mid l \in locVal_{\mathcal{K},s,\beta}(ld)\}$

Note that the RESTRICT constructor replaces the REJECT constructor of [13] based on the equivalency:

$$\text{REJECT}(u, u') \quad \equiv \quad \text{RESTRICT}(u, \texttt{everything} \setminus dom(u'))$$

for any two updates u and u'.

It is easy to see that $\equiv$ is a congruence relation with respect to the new constructors, too.

2.6.3 Meta-Level

To simplify the notation of rules in the following chapter, a new auxiliary set of syntactic expressions is introduced:

Definition 2.18 ($Alien_{obs}$).
Given a signature Σ, define the set
$Alien_{obs} := \{ (\{u\}g) \mid u \in Upd_\Sigma, g \in FSym_{obs}\}$

The arity function α is extended to include $Alien_{obs}$:

$$\alpha((\{u\}g)) := \alpha(g), \text{ for all } (\{u\}g) \in Alien_{obs}$$

In the following, expressions like $G(\bar{t})$ with $G \in FSym_{obs} \cup Alien_{obs}$ will be used, which means either $g(\bar{t})$ or $(\{u\}g)(\bar{t})$ with $g \in FSym_{obs}$, $u \in Upd_\Sigma$.

Chapter 3

Update Simplification Rules

In this chapter, the update simplification rules will be formulated. As mentioned before, the rewrite system is based on the system given in [13]. To give a complete picture, those rules are included in the following enumeration. Rules (R1)−(R80) stem from [13], rules from (NR81) upwards are new and a contribution of this thesis.[1] $fv(\alpha)$ denotes the set of free variables of expression α.

3.1 Updates and Location Descriptors

The rules in this section handle update application to all the different kinds of expression. This involves some rules which handle location descriptors, too. The overall goal is to get rid of updates as far as possible.

First of all, the following rules will show how sequential update composition $(u_1; u_2)$, the RESTRICT constructor and the *dom* constructor can be eliminated entirely. That is why these expressions will not appear in the rules following thereafter. Equally, substitutions are eliminated completely (as will be shown in section 3.2) and will not appear on the left hand side of rules in this section at all.

Sequential update composition is really only a convenience operation. It can be handled with exactly one rule:

---

sequential updates

$$u_1; u_2 \quad \rightsquigarrow \quad u_1 \| (\{u_1\} u_2) \qquad \text{(R45)}$$

[1] The old rules are partially adapted in the following way: some rules (like (R14)) still operate on the whole of $FSym$ (now including observer symbols, too) while others (like (R11)) are restricted to $FSym_{loc}$. Then, the REJECT constructor has been dropped in favour of the RESTRICT constructor, affecting rule (R64) and replacing (R22)−(R26) by (NR132)−(NR137). The syntax for non-recursive update application has changed, abandoning the NON-REC constructor (rules (R2) and (R10)−(R15)). Finally, the IN-DOM constructor has been dropped, replacing "IN-DOM$(f, \bar{t}, u)$" by "$locSubset(f(\bar{t}), dom(u))$" in rules (R13) and (R15) and making rules (R16)−(R21) obsolete.

RESTRICT can be eliminated completely in the presence of intersection on location descriptors:

RESTRICT

$$
\begin{array}{rcll}
\text{RESTRICT}(\texttt{skip}, ld) & \rightsquigarrow & \texttt{skip} & \text{(NR132)} \\
\text{RESTRICT}(u_1 \| u_2, ld) & \rightsquigarrow & \text{RESTRICT}(u_1, ld) & \text{(NR133)} \\
 & & \| \ \text{RESTRICT}(u_2, ld) & \\
\text{RESTRICT}(\texttt{if } \varphi.u, ld) & \rightsquigarrow & \texttt{if } \varphi.\text{RESTRICT}(u, ld) & \text{(NR134)} \\
\text{RESTRICT}(\texttt{for } x.u, ld) & \rightsquigarrow & \texttt{for } x.\text{RESTRICT}(u, ld) & \text{(NR135)} \\
 & & , x \notin fv(ld) & \\
\text{RESTRICT}(f(\bar{t}) := r, ld) & \rightsquigarrow & \texttt{if } \mathit{locSubset}(f(\bar{t}), ld).(f(\bar{t}) := r) & \text{(NR136)} \\
\text{RESTRICT}(ld' := *n, ld) & \rightsquigarrow & (ld' \cap ld) := *n & \text{(NR137)}
\end{array}
$$

For convenience, the following short-cut rules could be added:

$$
\begin{array}{rcl}
\text{RESTRICT}(u, \texttt{empty}) & \rightsquigarrow & \texttt{skip} \\
\text{RESTRICT}(u, \texttt{everything}) & \rightsquigarrow & u
\end{array}
$$

Note that there is no rule for the case $\text{RESTRICT}(\{u'\}u, ld)$, as $\{u'\}u$ can be simplified first, resulting in one of the cases above (appropriate rules will be given soon.)

The *dom* constructor can be eliminated in quite a straightforward way:

---

domain of updates

$$
\begin{array}{rcll}
\mathit{dom}(\texttt{skip}) & \rightsquigarrow & \texttt{empty} & \text{(NR103)} \\
\mathit{dom}(u_1 \| u_2) & \rightsquigarrow & \mathit{dom}(u_1) \| \mathit{dom}(u_2) & \text{(NR104)} \\
\mathit{dom}(\texttt{if } \varphi.u) & \rightsquigarrow & \texttt{if } \varphi.\mathit{dom}(u) & \text{(NR105)} \\
\mathit{dom}(\texttt{for } x.u) & \rightsquigarrow & \texttt{for } x.\mathit{dom}(u) & \text{(NR106)} \\
\mathit{dom}(f(\bar{t}) := r) & \rightsquigarrow & f(\bar{t}) & \text{(NR107)} \\
\mathit{dom}(ld := *n) & \rightsquigarrow & ld & \text{(NR108)}
\end{array}
$$

So now sequential updates, RESTRICT and *dom* are handled and do not have to be considered on the left hand side of rules in the following.

Let us turn to the application of updates. Update application is pushed down along the structure of terms, formulae, location descriptors and updates as far as possible:

---

update application on terms

$$
\begin{array}{rcll}
\{u\}x & \rightsquigarrow & x & \text{(R1)} \\
 & & , x \in \mathit{VSym} & \\
\{u\}h(\bar{t}) & \rightsquigarrow & (\{u\}h)(\{u\}\bar{t}) & \text{(R2)} \\
 & & , h \in \mathit{FSym} = \mathit{FSym}_{loc} \cup \mathit{FSym}_{obs} & \\
\{u\}(\{\texttt{everything} := *n\}f)(\bar{t}) & \rightsquigarrow & (\{\texttt{everything} := *n\}f)(\{u\}\bar{t}) & \text{(NR81)} \\
 & & , f \in \mathit{FSym}_{loc} & \\
\{u\}(\{u'\}g)(\bar{t}) & \rightsquigarrow & (\{u; u'\}g)(\{u\}\bar{t}) & \text{(NR82)} \\
 & & , g \in \mathit{FSym}_{obs} & \\
\{u\}\text{if } \varphi \text{ then } t_1 \text{ else } t_2 & \rightsquigarrow & \text{if } \{u\}\varphi \text{ then } \{u\}t_1 \text{ else } \{u\}t_2 & \text{(R3)} \\
\{u\}\text{min } x.\varphi & \rightsquigarrow & \text{min } x.\{u\}\varphi & \text{(R4)} \\
 & & , x \notin fv(u) &
\end{array}
$$

In rule (NR81), there is no need to include generic non-recursive updates (i. e. give a rule for $\{u\}(\{u'\}f)(\bar{t})$), as all updates but $\texttt{everything} := *n$ can be eliminated in front of location function symbols, as we will see presently.

---

update application on formulae

$\{u\}true$	$\rightsquigarrow$	$true$	(R5)
$\{u\}false$	$\rightsquigarrow$	$false$	(R5)
$\{u\}(\varphi_1 \wedge \varphi_2)$	$\rightsquigarrow$	$\{u\}\varphi_1 \wedge \{u\}\varphi_2$	(R6)
$\{u\}(\varphi_1 \vee \varphi_2)$	$\rightsquigarrow$	$\{u\}\varphi_1 \vee \{u\}\varphi_2$	(R6)
$\{u\}(\varphi_1 \rightarrow \varphi_2)$	$\rightsquigarrow$	$\{u\}\varphi_1 \rightarrow \{u\}\varphi_2$	(NR83)
$\{u\}\neg\varphi$	$\rightsquigarrow$	$\neg\{u\}\varphi$	(R7)
$\{u\}\forall x.\varphi$	$\rightsquigarrow$	$\forall x.\{u\}\varphi$	(R8)
		$, x \notin fv(u)$	
$\{u\}\exists x.\varphi$	$\rightsquigarrow$	$\exists x.\{u\}\varphi$	(R8)
		$, x \notin fv(u)$	
$\{u\}(t_1 \doteq t_2)$	$\rightsquigarrow$	$\{u\}t_1 \doteq \{u\}t_2$	(R9)
$\{u\}(t_1 \dot{<} t_2)$	$\rightsquigarrow$	$\{u\}t_1 \dot{<} \{u\}t_2$	(R9)
$\{u\}locEqual(ld_1, ld_2)$	$\rightsquigarrow$	$locEqual(\{u\}ld_1, \{u\}ld_2)$	(NR84)
$\{u\}locSubset(ld_1, ld_2)$	$\rightsquigarrow$	$locSubset(\{u\}ld_1, \{u\}ld_2)$	(NR85)
$\{u\}locDisjoint(ld_1, ld_2)$	$\rightsquigarrow$	$locDisjoint(\{u\}ld_1, \{u\}ld_2)$	(NR86)

---

update application on location descriptors

$\{u\}\texttt{empty}$	$\rightsquigarrow$	$\texttt{empty}$	(NR87)
$\{u\}\texttt{everything}$	$\rightsquigarrow$	$\texttt{everything}$	(NR88)
$\{u\}(ld_1 \| ld_2)$	$\rightsquigarrow$	$(\{u\}ld_1) \| (\{u\}ld_2)$	(NR89)
$\{u\}(\texttt{if}\ \varphi.ld)$	$\rightsquigarrow$	$\texttt{if}\ (\{u\}\varphi).(\{u\}ld)$	(NR90)
$\{u\}(\texttt{for}\ x.ld)$	$\rightsquigarrow$	$\texttt{for}\ x.(\{u\}ld)$	(NR91)
		$, x \notin fv(u)$	
$\{u\}f(\bar{t})$	$\rightsquigarrow$	$f(\{u\}\bar{t})$	(NR92)
		$, f \in FSym_{loc}$	
$\{u\}g(\bar{t})$	$\rightsquigarrow$	$(\{u\}g)(\{u\}\bar{t})$	(NR93)
		$, g \in FSym_{obs}$	
$\{u\}(\{u'\}g)(\bar{t})$	$\rightsquigarrow$	$(\{u; u'\}g)(\{u\}\bar{t})$	(NR94)
		$, g \in FSym_{obs}$	
$\{u\}(ld_1 \cap ld_2)$	$\rightsquigarrow$	$(\{u\}ld_1) \cap (\{u\}ld_2)$	(NR95)
$\{u\}(ld_1 \setminus ld_2)$	$\rightsquigarrow$	$(\{u\}ld_1) \setminus (\{u\}ld_2)$	(NR96)

The reason that there is no rule for $\{u\}(\{u'\}f)(\bar{t})$ with $f \in FSym_{loc}$ is, that the u' in front of the location function symbol f can be eliminated completely (by rule (NR100)).

update application on updates

$$
\begin{array}{rcll}
\{u\}\texttt{skip} & \rightsquigarrow & \texttt{skip} & \text{(R46)} \\
\{u\}(u_1 \| u_2) & \rightsquigarrow & (\{u\}u_1) \| (\{u\}u_2) & \text{(R48)} \\
\{u\}(\texttt{if } \varphi.u_1) & \rightsquigarrow & \texttt{if } (\{u\}\varphi).(\{u\}u_1) & \text{(R49)} \\
\{u\}(\texttt{for } x.u_1) & \rightsquigarrow & \texttt{for } x.(\{u\}u_1) & \text{(R50)} \\
 & & , x \notin fv(u) & \\
\{u\}(f(\bar{s}) := t) & \rightsquigarrow & f(\{u\}\bar{s}) := (\{u\}t) & \text{(R47)} \\
\{u\}(ld := *n) & \rightsquigarrow & (\{u\}ld) := *n & \text{(NR97)}
\end{array}
$$

Reaching function symbols (in rules (R2) and (NR93)), updates are applied non-recursively:

non-recursive update application on terms

$$
\begin{array}{rcll}
(\{\texttt{skip}\}h)(\bar{t}) & \rightsquigarrow & h(\bar{t}) & \text{(R10)} \\
 & & , h \in FSym = FSym_{loc} \cup FSym_{obs} & \\
(\{u_1 \| u_2\}f)(\bar{t}) & \rightsquigarrow & \text{if } locSubset(f(\bar{t}), dom(u_2)) & \text{(R13)} \\
 & & \text{then } (\{u_2\}f)(\bar{t}) & \\
 & & \text{else } (\{u_1\}f)(\bar{t}) & \\
 & & , f \in FSym_{loc} & \\
(\{\texttt{if } \varphi.u\}h)(\bar{t}) & \rightsquigarrow & \text{if } \varphi \text{ then } (\{u\}h)(\bar{t}) \text{ else } h(\bar{t}) & \text{(R14)} \\
 & & , h \in FSym = FSym_{loc} \cup FSym_{obs} & \\
(\{\texttt{for } x.u\}f)(\bar{t}) & \rightsquigarrow & (\{u[r/x]\}f)(\bar{t}) & \text{(R15)} \\
 & & , r = \min x.locSubset(f(\bar{t}), dom(u)) & \\
 & & , f \in FSym_{loc},\ x \notin fv(\bar{t}) & \\
(\{f(\bar{s}) := r\}f)(\bar{t}) & \rightsquigarrow & \text{if } \bar{t} \doteq \bar{s} \text{ then } r \text{ else } f(\bar{t}) & \text{(R11)} \\
 & & , f \in FSym_{loc} & \\
(\{f'(\bar{s}) := r\}f)(\bar{t}) & \rightsquigarrow & f(\bar{t}) & \text{(R12)} \\
 & & , f \in FSym_{loc},\ f \neq f' & \\
(\{ld := *n\}f)(\bar{t}) & \rightsquigarrow & \text{if } locSubset(f(\bar{t}), ld) & \text{(NR98)} \\
 & & \text{then } (\{\texttt{everything} := *n\}f)(\bar{t}) & \\
 & & \text{else } f(\bar{t}) & \\
 & & , f \in FSym_{loc},\ ld \neq \texttt{everything} &
\end{array}
$$

Note that the cases

- $(\{\texttt{everything} := *n\}f)(\bar{t})$, $f \in FSym_{loc}$
- $(\{u\}g)(\bar{t})$, $g \in FSym_{obs}$

are not covered by the rules above.

It is striking that almost all of the rules (except for (R10) and (R14)) are only applicable to location function symbols. Observer function symbols cannot be updated by a local rewriting system in a generic way (this effect will be examined more closely in chapter 6).

The application of updated updates (i. e. the case $(\{\{u'\}u\}h)(\bar{t})$) does not need to be handled explicitly, as $\{u'\}u$ can be reduced to a "flat" update by the rules given before.

For location descriptors we have to handle non-recursive update application at some point, too:

---

non-recursive update application on location descriptors

$$\begin{array}{rcll} (\{u\}f)(\bar{t}) & \rightsquigarrow & f(\bar{t}) & \text{(NR100)} \\ & & , f \in FSym_{loc} & \\ (\{\texttt{skip}\}g)(\bar{t}) & \rightsquigarrow & g(\bar{t}) & \text{(NR101)} \\ & & , g \in FSym_{obs} & \end{array}$$

Updates applied (non-recursively) to location function symbols ($(\{u\}f)(\bar{t})$ with $f \in FSym_{loc}$) can simply be dropped, as they do not affect the semantics of the location descriptor (confer section 2.6.2).

As for updates applied to observer function symbols in location descriptors ($(\{u\}g)(\bar{t})$ with $g \in FSym_{obs}$), there is not much we can do. The reason is that location descriptors lack framing: the dependencies of observers can change from state to state arbitrarily, there is no restriction on the *depends* function which fixes the dependencies for each state separately (see definition 2.6). Thus, having a location descriptor like $(\{u\}g)(\bar{t})$, there is no generic way to express the same set of locations in another state than the one reachable by the update u and so the expression cannot be simplified unless u is the empty update (except for simplifications inside of u or $\bar{t}$ of course).

---

anonymising updates

$$\texttt{empty} := *n \quad \rightsquigarrow \quad \texttt{skip} \qquad \text{(NR102)}$$

One could think of mapping location descriptor composition in anonymising updates to update composition using the following rules:

$$\begin{array}{rcl} (ld_1 \| ld_2) := *n & \rightsquigarrow & (ld_1 := *n) \| (ld_2 := *n) \\ (\texttt{if } \varphi.ld) := *n & \rightsquigarrow & \texttt{if } \varphi.(ld := *n) \\ (\texttt{for } x.ld) := *n & \rightsquigarrow & \texttt{for } x.(ld := *n) \end{array}$$

However, composed location descriptors are easier to handle than composed updates in general. This is because parallel ($u_1 \| u_2$) and quantified updates ($\texttt{for } \varphi.u$) require clash resolution (as described in section 2.3), while parallel and quantified location descriptors do not.

Simplifying, for instance, the application of an anonymous update containing a quantified location descriptor, like

$$(\{(\texttt{for } x.obs(x)) := *3\}f)(t)$$

results in

$$\begin{array}{l} \text{if } \exists x.locSubset(f(t), obs(x)) \\ \text{then } (\{\texttt{everything} := *3\}f)(t) \\ \text{else } f(t) \end{array}$$

which involves the condition $\exists x.locSubset(f(t), obs(x))$ where any witness for x such that $locSubset(f(t), obs(x))$ is sufficient.

In contrast to that, pulling the quantification to the update level:

$(\{\texttt{for}\ x.(obs(x) := *3)\}f)(t)$

results in

if $locSubset(f(t), obs(\min\ x.locSubset(f(t), obs(x))))$
then $(\{\texttt{everything} := *3\}f)(t)$
else $f(t)$

with a slightly more complex condition containing the min constructor which selects the minimal x such that the *locSubset* expression holds (if such an x exists at all).

As a matter of fact, complex location descriptors can remain in anonymising updates anyway due to the intersection and difference operators.

Though not in the focus of the update simplifier, *locSubset* expressions are simplified, too. Before giving the rules, a special condition for two of them should be pointed out. Rules (NR118) and (NR119) below are sound only if the set of locations $Locations_{\mathcal{K}}$ (see definition 2.8) satisfies $Locations_{\mathcal{K}} \neq \emptyset$ or $|Locations_{\mathcal{K}}| \geq 2$ respectively, for *all* Kripke structures $\mathcal{K}$ which are valid in the current context. These conditions are implied, for example, by $|FSym_{loc}| \geq 2$, which is usually the case.

---

locSubset

$locSubset(\texttt{empty}, ld)$	$\rightsquigarrow$	$true$	(NR109)
$locSubset(ld_1 \| ld_2, ld)$	$\rightsquigarrow$	$locSubset(ld_1, ld)$ $\wedge\ locSubset(ld_2, ld)$	(NR110)
$locSubset(\texttt{if}\ \varphi.ld', ld)$	$\rightsquigarrow$	$\neg\varphi \vee locSubset(ld', ld)$	(NR111)
$locSubset(\texttt{for}\ x.ld', ld)$	$\rightsquigarrow$	$\forall x.locSubset(ld', ld)$ $, x \notin fv(ld)$	(NR112)
$locSubset(ld_1 \setminus ld_2, ld)$	$\rightsquigarrow$	$locSubset(ld_1, ld \| ld_2)$	(NR113)
$locSubset(ld, \texttt{everything})$	$\rightsquigarrow$	$true$	(NR114)
$locSubset(ld, \texttt{if}\ \varphi.ld')$	$\rightsquigarrow$	$\varphi \wedge locSubset(ld, ld')$ $\vee\ locSubset(ld, \texttt{empty})$	(NR115)
$locSubset(ld, ld_1 \cap ld_2)$	$\rightsquigarrow$	$locSubset(ld, ld_1)$ $\wedge\ locSubset(ld, ld_2)$	(NR116)
$locSubset(ld, ld_1 \setminus ld_2)$	$\rightsquigarrow$	$locSubset(ld, ld_1)$ $\wedge\ locDisjoint(ld, ld_2)$	(NR117)
$locSubset(\texttt{everything}, \texttt{empty})$	$\rightsquigarrow$	$false$ $, Locations_{\mathcal{K}} \neq \emptyset$	(NR118)
$locSubset(\texttt{everything}, f(\bar{t}))$	$\rightsquigarrow$	$false$ $, f \in FSym_{loc},\ \|Locations_{\mathcal{K}}\| \geq 2$	(NR119)

$locSubset(f(\bar{t}), ld) \rightsquigarrow \ldots\ [f \in FSym_{loc}]$:

$locSubset(f(\bar{t}), \texttt{empty})$	$\rightsquigarrow$	$false$	(NR120)
$locSubset(f(\bar{t}), ld_1 \| ld_2)$	$\rightsquigarrow$	$locSubset(f(\bar{t}), ld_1)$ $\vee\ locSubset(f(\bar{t}), ld_2)$	(NR121)
$locSubset(f(\bar{t}), \texttt{for}\ x.ld)$	$\rightsquigarrow$	$\exists x.locSubset(f(\bar{t}), ld)$ $, x \notin fv(\bar{t})$	(NR122)
$locSubset(f(\bar{t}), f(\bar{s}))$	$\rightsquigarrow$	$\bar{t} \doteq \bar{s}$	(NR123)
$locSubset(f(\bar{t}), f'(\bar{s}))$	$\rightsquigarrow$	$false$ $, f' \in FSym_{loc}, f \neq f'$	(NR124)

Here, the following expressions are unhandled:

- $locSubset(f(\bar{t}), G(\bar{s}))$, $f \in FSym_{loc}$, $G \in FSym_{obs} \cup Alien_{obs}$
- $locSubset(\texttt{everything}, ld)$
  where $ld \in \{\ G(\bar{t}),\ ld_1 \| ld_2,\ \texttt{for}\ x.ld'\ \}$
  with $G \in FSym_{obs} \cup Alien_{obs}$
- $locSubset(G_1(\bar{t}_1) \cap \ldots \cap G_n(\bar{t}_n), ld)$, $n \geq 1$, $G_i \in FSym_{obs} \cup Alien_{obs}$
  where $ld \in \{\ f(\bar{t}),\ G'(\bar{s}),\ \texttt{empty},\ ld_1 \| ld_2,\ \texttt{for}\ x.ld'\ \}$
  with $f \in FSym_{loc}$, $G' \in FSym_{obs} \cup Alien_{obs}$

The third case ($locSubset(G_1(\bar{t}_1) \cap \ldots \cap G_n(\bar{t}_n), ld)$) looks a bit odd at first sight. One might argue, that the unhandled cases are simply $locSubset(G(\bar{t}), ld)$ and $locSubset(ld_1 \cap ld_2, ld)$. Yet, as we will see, intersection can be eliminated up to $G_1(\bar{t}_1) \cap \ldots \cap G_n(\bar{t}_n)$.

locEqual and *locDisjoint* are mapped to *locSubset*:

---

locEqual

$$locEqual(ld_1, ld_2) \quad \rightsquigarrow \quad locSubset(ld_1, ld_2) \wedge locSubset(ld_2, ld_1) \qquad \text{(NR130)}$$

---

locDisjoint

$$locDisjoint(ld_1, ld_2) \quad \rightsquigarrow \quad locSubset(ld_1 \cap ld_2, \texttt{empty}) \qquad \text{(NR131)}$$

Trying to simplify location descriptors, the following rules handle the intersection and difference operators:

---

intersection of location descriptors

$$\begin{array}{rcll}
ld \cap \texttt{empty} & \rightsquigarrow & \texttt{empty} & \text{(NR138)} \\
ld \cap \texttt{everything} & \rightsquigarrow & ld & \text{(NR139)} \\
ld \cap (ld_1 \| ld_2) & \rightsquigarrow & (ld \cap ld_1) \| (ld \cap ld_2) & \text{(NR140)} \\
ld \cap \texttt{if}\ \varphi.ld' & \rightsquigarrow & \texttt{if}\ \varphi.(ld \cap ld') & \text{(NR141)} \\
ld \cap \texttt{for}\ x.ld' & \rightsquigarrow & \texttt{for}\ x.(ld \cap ld') & \text{(NR142)} \\
 & & , x \notin fv(ld) & \\
ld \cap (ld_1 \setminus ld_2) & \rightsquigarrow & (ld \cap ld_1) \setminus ld_2 & \text{(NR143)} \\
f(\bar{s}) \cap f(\bar{t}) & \rightsquigarrow & \texttt{if}\ (\bar{s} \doteq \bar{t}).f(\bar{t}) & \text{(NR144)} \\
 & & , f \in FSym_{loc} & \\
f'(\bar{s}) \cap f(\bar{t}) & \rightsquigarrow & \texttt{empty} & \text{(NR145)} \\
 & & , f, f' \in FSym_{loc},\ f \neq f' & \\
(G_1(\bar{s}_1) \cap \ldots \cap G_n(\bar{s}_n)) \cap f(\bar{t}) & \rightsquigarrow & & \\
 & & \texttt{if}\ locSubset(f(\bar{t}), G_1(\bar{s}_1) \cap \ldots \cap G_n(\bar{s}_n)).f(\bar{t}) & \text{(NR146)} \\
 & & , f \in FSym_{loc},\ n \geq 1 & \\
 & & , G_i \in FSym_{obs} \cup Alien_{obs} &
\end{array}$$

The intersection operator is symmetric. Accordingly, the symmetric cases of the rules above should be included, too, or symmetry could be handled in some other fashion.

The following case is not covered by the given rules:

- $G_1(\bar{s}_1) \cap \ldots \cap G_n(\bar{s}_n)$, $n \geq 2$, $G_i \in FSym_{obs} \cup Alien_{obs}$

The expression $G_1(\bar{s}_1) \cap \ldots \cap G_n(\bar{s}_n)$ makes use of the associativity of $\cap$ and thus any parentheses are omitted.

Finally, it should be pointed out, that intersection is redundant with difference and the `everything` location descriptor and hence the following rule could be used to handle intersection:

$$ld_1 \cap ld_2 \quad \rightsquigarrow \quad ld_1 \setminus (\texttt{everything} \setminus ld_2)$$

However, the calculus is not good at handling the resulting expression and therefore the specialised rules given above are used.

---

difference of location descriptors

$ld \setminus ld$	$\rightsquigarrow$	`empty`	(NR150)
`empty` $\setminus\, ld$	$\rightsquigarrow$	`empty`	(NR151)
$(ld_1 \| ld_2) \setminus ld$	$\rightsquigarrow$	$(ld_1 \setminus ld) \| (ld_2 \setminus ld)$	(NR152)
(`if` $\varphi.ld') \setminus ld$	$\rightsquigarrow$	`if` $\varphi.(ld' \setminus ld)$	(NR153)
(`for` $x.ld') \setminus ld$	$\rightsquigarrow$	`for` $x.(ld' \setminus ld)$, $x \notin fv(ld)$	(NR154)
$f(\bar{t}) \setminus ld$	$\rightsquigarrow$	`if` $\neg locSubset(f(\bar{t}), ld).f(\bar{t})$, $f \in FSym_{loc}$	(NR155)
$ld \setminus$ `empty`	$\rightsquigarrow$	ld	(NR156)
$ld \setminus$ `everything`	$\rightsquigarrow$	`empty`	(NR157)
$ld \setminus (ld_1 \| ld_2)$	$\rightsquigarrow$	$(ld \setminus ld_1) \setminus ld_2$	(NR158)
$ld \setminus$ `if` $\varphi.ld'$	$\rightsquigarrow$	(`if` $\varphi.(ld \setminus ld'))\|$(`if` $\neg\varphi.ld$)	(NR159)

The following cases are not covered:

- $ld_1 \setminus ld_2$
  with $ld_1 \in \{$ `everything`, $G(\bar{s})$, $G_1(\bar{s}_1) \cap \ldots \cap G_n(\bar{s}_n)$, $ld'_1 \setminus ld'_2$ $\}$,
  $ld_2 \in \{$ `for` $x.ld$, $f(\bar{t})$, $G(\bar{s})$, $G_1(\bar{s}_1) \cap \ldots \cap G_n(\bar{s}_n)$, $ld'_1 \setminus ld'_2$ $\}$,
  where, for both ld_1 and ld_2 independently: $n \geq 2$, $f \in FSym_{loc}$, G,$G_i \in FSym_{obs} \cup Alien_{obs}$ and ld'_1, ld'_2 are of the same kind as ld_1, ld_2 respectively

Several simplification rules can be defined:

---

simplification rules based on neutral and extremal elements

`if` φ`.skip`	$\rightsquigarrow$	`skip`	(R66)
`if` *false*.u	$\rightsquigarrow$	`skip`	(R67)
`if` *true*.u	$\rightsquigarrow$	u	(R68)
`for` x`.skip`	$\rightsquigarrow$	`skip`	(R69)
`for` $x.u$	$\rightsquigarrow$	u , $x \notin fv(u)$	(R70)
`skip`$\|u$	$\rightsquigarrow$	u	(R71)
$u\|$`skip`	$\rightsquigarrow$	u	(R72)
`skip`; u	$\rightsquigarrow$	u	(R73)
u; `skip`	$\rightsquigarrow$	u	(R74)

$$
\begin{array}{rcll}
u \| (\texttt{everything} := *n) & \rightsquigarrow & \texttt{everything} := *n & \text{(NR160)} \\
(\texttt{everything} := *n) \| (ld := *n) & \rightsquigarrow & \texttt{everything} := *n & \text{(NR161)} \\
\{\texttt{skip}\}t & \rightsquigarrow & t & \text{(NR162)} \\
\{\texttt{skip}\}\varphi & \rightsquigarrow & \varphi & \text{(NR163)} \\
\{\texttt{skip}\}u & \rightsquigarrow & u & \text{(NR164)} \\
\{\texttt{skip}\}ld & \rightsquigarrow & ld & \text{(NR165)} \\
 & & & \\
\texttt{empty} \| ld & \rightsquigarrow & ld & \text{(NR166)} \\
ld \| \texttt{empty} & \rightsquigarrow & ld & \text{(NR167)} \\
\texttt{everything} \| ld & \rightsquigarrow & \texttt{everything} & \text{(NR168)} \\
ld \| \texttt{everything} & \rightsquigarrow & \texttt{everything} & \text{(NR169)} \\
\texttt{if } \varphi.\texttt{empty} & \rightsquigarrow & \texttt{empty} & \text{(NR170)} \\
\texttt{if } \mathit{true}.ld & \rightsquigarrow & ld & \text{(NR171)} \\
\texttt{if } \mathit{false}.ld & \rightsquigarrow & \texttt{empty} & \text{(NR172)} \\
\texttt{for } x.ld & \rightsquigarrow & ld & \text{(NR173)} \\
 & & , x \notin fv(ld) & \\
\texttt{for } x.\texttt{empty} & \rightsquigarrow & \texttt{empty} & \text{(NR174)} \\
\texttt{for } x.\texttt{everything} & \rightsquigarrow & \texttt{everything} & \text{(NR175)} \\
\texttt{everything} \setminus (\texttt{everything} \setminus ld) & \rightsquigarrow & ld & \text{(NR176)}
\end{array}
$$

---

collecting rules

$$
\begin{array}{rcll}
\{u_1\}\{u_2\}t & \rightsquigarrow & \{u_1; u_2\}t & \text{(R51)} \\
\{u_1\}\{u_2\}\varphi & \rightsquigarrow & \{u_1; u_2\}\varphi & \text{(R51)} \\
\{u_1\}\{u_2\}u & \rightsquigarrow & \{u_1; u_2\}u & \text{(R51)} \\
\{u_1\}\{u_2\}ld & \rightsquigarrow & \{u_1; u_2\}ld & \text{(R51)} \\
 & & & \\
\texttt{if } \varphi.\texttt{if } \psi.u & \rightsquigarrow & \texttt{if } (\varphi \wedge \psi).u & \text{(R59)} \\
\texttt{if } \varphi.\texttt{if } \psi.ld & \rightsquigarrow & \texttt{if } (\varphi \wedge \psi).ld & \text{(NR177)}
\end{array}
$$

A couple of update normalisation rules are defined in [13]:

---

normalisation rules

$$
\begin{array}{rcll}
u_1 \| (u_2 \| u_3) & \rightsquigarrow & (u_1 \| u_2) \| u_3 & \text{(R52)} \\
\texttt{if } \varphi.(u_1 \| u_2) & \rightsquigarrow & (\texttt{if } \varphi.u_1) \| (\texttt{if } \varphi.u_2) & \text{(R58)} \\
\texttt{if } \varphi.\texttt{for } x.u & \rightsquigarrow & \texttt{for } x.\texttt{if } \varphi.u & \text{(R60)} \\
 & & , x \notin fv(\varphi) & \\
\texttt{for } x.(u_1 \| u_2) & \rightsquigarrow & (\texttt{for } x.u_1) & \text{(R64)} \\
 & & \| \ (\texttt{for } x.\mathrm{RESTRICT}(u_2, \texttt{everything} \setminus dom(u))) & \\
 & & , u = \texttt{for } z.\texttt{if } (z \dot{<} x).(u_1[z/x]) & \\
 & & , z \neq x,\ z \notin fv(u_1) &
\end{array}
$$

Similar rules could be defined for location descriptors. It is not so easy, however, to define a suitable normal form for them, as in different situations different forms are advantageous.

Therefore, respective transformations for location descriptors will be given as (non-directed) equivalencies a bit later.

3.2 Substitutions

Apart from update application and simplification, the rewrite system covers explicit substitution application. In fact, this part can be considered as a separate calculus in its own right, as all substitutions are eventually applied. For the sake of completeness, the respective rule set is given here:

---

substitutions on terms

$$
\begin{array}{rcll}
x[r/x] & \rightsquigarrow & r & \text{(R27)} \\
y[r/x] & \rightsquigarrow & y & \text{(R28)} \\
 & & , x \neq y,\ y \in \mathit{VSym} & \\
h(\bar{t})[r/x] & \rightsquigarrow & h(\bar{t}[r/x]) & \text{(R29)} \\
 & & , h \in \mathit{FSym} = \mathit{FSym}_{loc} \cup \mathit{FSym}_{obs} & \\
(\{\texttt{everything} := *n\}f)(\bar{t})[r/x] & \rightsquigarrow & & \\
 & & (\{\texttt{everything} := *n\}f)(\bar{t}[r/x]) & \text{(NR178)} \\
 & & , f \in \mathit{FSym}_{loc} & \\
(\{u\}g)(\bar{t})[r/x] & \rightsquigarrow & (\{u[r/x]\}g)(\bar{t}[r/x]) & \text{(NR179)} \\
 & & , g \in \mathit{FSym}_{obs} & \\
(\text{if } \varphi \text{ then } t_1 \text{ else } t_2)[r/x] & \rightsquigarrow & \text{if } \varphi[r/x] & \text{(R30)} \\
 & & \text{then } t_1[r/x] & \\
 & & \text{else } t_2[r/x] & \\
(\min x.\varphi)[r/x] & \rightsquigarrow & \min x.\varphi & \text{(R31)} \\
(\min y.\varphi)[r/x] & \rightsquigarrow & \min y.(\varphi[r/x]) & \text{(R32)} \\
 & & , x \neq y,\ y \notin fv(r) &
\end{array}
$$

---

substitutions on formulae

$$
\begin{array}{rcll}
\mathit{true}[r/x] & \rightsquigarrow & \mathit{true} & \text{(R33)} \\
\mathit{false}[r/x] & \rightsquigarrow & \mathit{false} & \text{(R33)} \\
(\varphi_1 \wedge \varphi_2)[r/x] & \rightsquigarrow & \varphi_1[r/x] \wedge \varphi_2[r/x] & \text{(R34)} \\
(\varphi_1 \vee \varphi_2)[r/x] & \rightsquigarrow & \varphi_1[r/x] \vee \varphi_2[r/x] & \text{(R34)} \\
(\varphi_1 \rightarrow \varphi_2)[r/x] & \rightsquigarrow & \varphi_1[r/x] \rightarrow \varphi_2[r/x] & \text{(NR180)} \\
(\neg\varphi)[r/x] & \rightsquigarrow & \neg(\varphi[r/x]) & \text{(R35)} \\
(\forall x.\varphi)[r/x] & \rightsquigarrow & \forall x.\varphi & \text{(R36)} \\
(\exists x.\varphi)[r/x] & \rightsquigarrow & \exists x.\varphi & \text{(R36)} \\
(\forall y.\varphi)[r/x] & \rightsquigarrow & \forall y.(\varphi[r/x]) & \text{(R37)} \\
 & & , x \neq y,\ y \notin fv(r) & \\
(\exists y.\varphi)[r/x] & \rightsquigarrow & \exists y.(\varphi[r/x]) & \text{(R37)} \\
 & & , x \neq y,\ y \notin fv(r) & \\
(t_1 \doteq t_2)[r/x] & \rightsquigarrow & t_1[r/x] \doteq t_2[r/x] & \text{(R38)} \\
(t_1 \dot{<} t_2)[r/x] & \rightsquigarrow & t_1[r/x] \dot{<} t_2[r/x] & \text{(R38)} \\
\mathit{locEqual}(ld_1, ld_2)[r/x] & \rightsquigarrow & \mathit{locEqual}(ld_1[r/x], ld_2[r/x]) & \text{(NR181)} \\
\mathit{locSubset}(ld_1, ld_2)[r/x] & \rightsquigarrow & \mathit{locSubset}(ld_1[r/x], ld_2[r/x]) & \text{(NR182)} \\
\mathit{locDisjoint}(ld_1, ld_2)[r/x] & \rightsquigarrow & \mathit{locDisjoint}(ld_1[r/x], ld_2[r/x]) & \text{(NR183)}
\end{array}
$$

substitutions on location descriptors

$$
\begin{array}{rcll}
\texttt{empty}[r/x] & \rightsquigarrow & \texttt{empty} & \text{(NR184)} \\
\texttt{everything}[r/x] & \rightsquigarrow & \texttt{everything} & \text{(NR185)} \\
(ld_1 \| ld_2)[r/x] & \rightsquigarrow & (ld_1[r/x]) \| (ld_2[r/x]) & \text{(NR186)} \\
(\texttt{if } \varphi.ld)[r/x] & \rightsquigarrow & \texttt{if } (\varphi[r/x]).(ld[r/x]) & \text{(NR187)} \\
(\texttt{for } x.ld)[r/x] & \rightsquigarrow & \texttt{for } x.ld & \text{(NR188)} \\
(\texttt{for } y.ld)[r/x] & \rightsquigarrow & \texttt{for } y.(ld[r/x]) & \text{(NR189)} \\
 & & , y \neq x,\ y \notin fv(r) & \\
h(\bar{t})[r/x] & \rightsquigarrow & h(\bar{t}[r/x]) & \text{(NR190)} \\
 & & , h \in FSym = FSym_{loc} \cup FSym_{obs} & \\
((\{u\}g)(\bar{t}))[r/x] & \rightsquigarrow & (\{u[r/x]\}g)(\bar{t}[r/x]) & \text{(NR191)} \\
 & & , g \in FSym_{obs} & \\
(ld_1 \cap ld_2)[r/x] & \rightsquigarrow & (ld_1[r/x]) \cap (ld_2[r/x]) & \text{(NR192)} \\
(ld_1 \setminus ld_2)[r/x] & \rightsquigarrow & (ld_1[r/x]) \setminus (ld_2[r/x]) & \text{(NR193)}
\end{array}
$$

substitutions on updates

$$
\begin{array}{rcll}
\texttt{skip}[r/x] & \rightsquigarrow & \texttt{skip} & \text{(R39)} \\
(u_1 \| u_2)[r/x] & \rightsquigarrow & (u_1[r/x]) \| (u_2[r/x]) & \text{(R41)} \\
(\texttt{if } \varphi.u)[r/x] & \rightsquigarrow & \texttt{if } (\varphi[r/x]).(u[r/x]) & \text{(R42)} \\
(\texttt{for } x.u)[r/x] & \rightsquigarrow & \texttt{for } x.u & \text{(R43)} \\
(\texttt{for } y.u)[r/x] & \rightsquigarrow & \texttt{for } y.(u[r/x]) & \text{(R44)} \\
 & & , x \neq y,\ y \notin fv(r) & \\
(f(\bar{t}) := s)[r/x] & \rightsquigarrow & f(\bar{t}[r/x]) := s[r/x] & \text{(R40)} \\
(ld := *n)[r/x] & \rightsquigarrow & (ld[r/x]) := *n & \text{(NR194)}
\end{array}
$$

3.3 Heuristic Rules

Defining the rule set above, we came across different expressions, which could not be simplified in a generic way like the other ones. Examples are non-recursive update application on observers ($(\{u\}g)(\bar{t}),\ g \in FSym_{obs}$) or certain *locSubset* expressions (e. g. $locSubset(g(\bar{t}), ld_1 \| ld_2),\ g \in FSym_{obs}$). Nonetheless, we *can* define rules for these expressions which prove to be useful. These rules differ from the ones given up to now by not succeeding to simplify the given expression in all cases. Sometimes subexpressions are added which turn out not to give any advantage or the resulting expressions are subsequently reduced to the expression we started with. This is another property common to all of these rules: their left hand side is either directly part of their right hand side or the right hand side will sometimes reduce to the left hand side again. That means that these rules are applicable infinitely without any advancement. Therefore, their application has to be kept track of, applying each rule at most once for any given expression. To stress this fact, the rules are written as follows: "$lhs \overset{1}{\rightsquigarrow} rhs$".

Given these properties, it is a good idea to apply these rules as late as possible, i. e. when no "regular" rule is applicable any more. Apart from all that, the following rules are able to simplify the previously irreducible expressions in various cases.

The first case we will consider is the non-recursive update application on terms:

$$(\{u\}g)(\bar{t}) \overset{1}{\rightsquigarrow} (\ \{\texttt{if}\ \neg \mathit{locDisjoint}(\mathit{dom}(u), g(\bar{t})).u\}\ g)(\bar{t}) \quad \text{(NR99)}$$
$$, g \in \mathit{FSym}_{obs}$$

We will have a closer look at this rule in section 6.2. It should be pointed out that this rule does not apply if $(\{u\}g)(\bar{t})$ is a location descriptor.

Now, let us consider the *locSubset* example given above: $\mathit{locSubset}(g(\bar{t}), ld_1 \| ld_2)$, $g \in \mathit{FSym}_{obs}$. There is no generic way to simplify this expression. If, for instance, ld_1 and ld_2 were two elementary location descriptors with location symbols (rendering the expression $\mathit{locSubset}(g(\bar{t}), f_1(\bar{t}_1) \| f_2(\bar{t}_2))$ with $g \in \mathit{FSym}_{obs}$, f_1,$f_2 \in \mathit{FSym}_{loc}$) we could not simplify the formula without further knowledge of $g(\bar{t})$'s dependencies. If, on the other hand, ld_1 was $g(\bar{t})$, too, then clearly $\mathit{locSubset}(g(\bar{t}), g(\bar{t}) \| ld_2)$ would be true.

This brings us to consider implications we could exploit to establish respective simplification rules:

$$\begin{aligned}
\mathit{locSubset}(G(\bar{t}), ld_1) &\Rightarrow \mathit{locSubset}(G(\bar{t}), ld_1 \| ld_2) \\
&\quad , G \in \mathit{FSym}_{obs} \cup \mathit{Alien}_{obs} \\
\exists x.\mathit{locSubset}(G(\bar{t}), ld) &\Rightarrow \mathit{locSubset}(G(\bar{t}), \texttt{for}\ x.ld) \\
&\quad , G \in \mathit{FSym}_{obs} \cup \mathit{Alien}_{obs},\ x \notin fv(G(\bar{t})) \\
\bar{t} \doteq \bar{s} &\Rightarrow \mathit{locSubset}(G(\bar{t}), G(\bar{s})) \\
&\quad , G \in \mathit{FSym}_{obs} \cup \mathit{Alien}_{obs}
\end{aligned}$$

We turn these into the following rules:

$$\begin{aligned}
\mathit{locSubset}(G(\bar{t}), ld_1 \| ld_2) &\overset{1}{\rightsquigarrow} \mathit{locSubset}(G(\bar{t}), ld_1) \quad \text{(NR127)} \\
&\quad \vee \mathit{locSubset}(G(\bar{t}), ld_2) \\
&\quad \vee \mathit{locSubset}(G(\bar{t}), ld_1 \| ld_2) \\
\mathit{locSubset}(G(\bar{t}), \texttt{for}\ x.ld) &\overset{1}{\rightsquigarrow} \exists x.\mathit{locSubset}(G(\bar{t}), ld) \quad \text{(NR128)} \\
&\quad \vee \mathit{locSubset}(G(\bar{t}), \texttt{for}\ x.ld) \\
&\quad , x \notin fv(G(\bar{t})) \\
\mathit{locSubset}(G(\bar{t}), G(\bar{s})) &\overset{1}{\rightsquigarrow} \bar{t} \doteq \bar{s} \quad \text{(NR129)} \\
&\quad \vee \mathit{locSubset}(G(\bar{t}), G(\bar{s}))
\end{aligned}$$

where $G \in \mathit{FSym}_{obs} \cup \mathit{Alien}_{obs}$ in all cases.

$$\begin{aligned}
&\mathit{locSubset}(\texttt{everything}, ld_1 \| ld_2) \\
&\qquad \overset{1}{\rightsquigarrow} \mathit{locSubset}(\texttt{everything}, ld_1) \quad \text{(NR125)} \\
&\qquad\quad \vee \mathit{locSubset}(\texttt{everything}, ld_2) \\
&\qquad\quad \vee \mathit{locSubset}(\texttt{everything}, ld_1 \| ld_2) \\
&\mathit{locSubset}(\texttt{everything}, \texttt{for}\ x.ld) \\
&\qquad \overset{1}{\rightsquigarrow} \exists x.\mathit{locSubset}(\texttt{everything}, ld) \quad \text{(NR126)} \\
&\qquad\quad \vee \mathit{locSubset}(\texttt{everything}, \texttt{for}\ x.ld)
\end{aligned}$$

For the intersection of location descriptors the implications

$$\begin{aligned}
\mathit{locDisjoint}(ld_1, ld_2) &\Rightarrow ld_1 \cap ld_2 \equiv \texttt{empty} \\
\mathit{locSubset}(ld_1, ld_2) &\Rightarrow ld_1 \cap ld_2 \equiv ld_1
\end{aligned}$$

lead to the rules:

$$G(\bar{s}) \cap G(\bar{t}) \overset{1}{\rightsquigarrow} \texttt{if } (\bar{s} \doteq \bar{t}).(G(\bar{s})) \parallel \texttt{if } (\neg\ \bar{s} \doteq \bar{t}).(G(\bar{s}) \cap G(\bar{t})) \quad \text{(NR147)}$$
$$, G \in FSym_{obs} \cup Alien_{obs}$$

$$G(\bar{s}) \cap G'(\bar{t}) \overset{1}{\rightsquigarrow} \texttt{if } \neg locDisjoint(G(\bar{s}), G'(\bar{t})).(G(\bar{s}) \cap G'(\bar{t})) \quad \text{(NR148)}$$
$$, G,G' \in FSym_{obs} \cup Alien_{obs}$$

$$G(\bar{s}) \cap G'(\bar{t}) \overset{1}{\rightsquigarrow} \texttt{if } locSubset(G(\bar{s}), G'(\bar{t})).(G(\bar{s})) \quad \text{(NR149)}$$
$$\parallel \texttt{if } \neg locSubset(G(\bar{s}), G'(\bar{t})).(G(\bar{s}) \cap G'(\bar{t}))$$
$$, G,G' \in FSym_{obs} \cup Alien_{obs}$$

As these rules make the expressions notably more complex, they will probably be left for manual application. Alternatively a backtracking mechanism could be used.

Obviously, rule (NR149) has a symmetric counterpart where the subset relation is the other way around.

3.4 Useful Equivalencies

To complete the picture, here are several equivalencies, which do not come along as rewrite rules but rather can be applied in either direction as needed. Note that [13] lists further equivalencies (rules (R51)−(R65)) which are omitted here.

---

some useful equivalencies

$$\begin{aligned}
u &\equiv u \parallel \mathrm{RESTRICT}(u, ld) \\
u &\equiv \mathrm{RESTRICT}(u, ld) \parallel u \\
\texttt{if } \varphi.(ld_1 \| ld_2) &\equiv (\texttt{if } \varphi.ld_1) \parallel (\texttt{if } \varphi.ld_2) \\
\texttt{for } x.\texttt{if } \varphi.ld &\equiv \texttt{if } \varphi.\texttt{for } x.ld \\
&\quad , x \notin fv(\varphi) \\
\texttt{for } x.\texttt{if } \varphi.ld &\equiv \texttt{if } (\exists x.\varphi).ld \\
&\quad , x \notin fv(ld) \\
\texttt{for } x.(ld_1 \| ld_2) &\equiv (\texttt{for } x.ld_1) \parallel (\texttt{for } x.ld_2) \\
\texttt{for } x.\texttt{for } y.ld &\equiv \texttt{for } y.\texttt{for } x.ld \\
(ld \setminus ld_1) \setminus ld_2 &\equiv (ld \setminus ld_2) \setminus ld_1 \\
ld \setminus (ld_1 \setminus ld_2) &\equiv (ld \setminus ld_1) \parallel (ld \cap ld_2) \\
ld \setminus (ld \cap ld') &\equiv ld \setminus ld' \\
(ld_1 \cap ld) \setminus (ld_2 \cap ld) &\equiv (ld_1 \cap ld) \setminus ld_2 \\
ld_1 \| ld_2 &\equiv ld_2 \| ld_1 \\
ld_1 \parallel (ld_2 \| ld_3) &\equiv (ld_1 \| ld_2) \parallel ld_3 \\
ld &\equiv ld \parallel \texttt{if } \varphi.ld \\
ld_1 \cap ld_2 &\equiv ld_2 \cap ld_1 \\
ld_1 \cap (ld_2 \cap ld_3) &\equiv (ld_1 \cap ld_2) \cap ld_3
\end{aligned}$$

Chapter 4

Examples

A few examples will now demonstrate how the calculus works.

First of all, recall that the update simplifier works locally, while dependencies of observers are usually defined by axioms, i. e. globally. For reasoning at an abstract level we need to combine local and global operations. In the following examples we will apply the axioms "manually", using the fact that *locEqual* is the equality on the set of location descriptors. By its definition (see definition 2.14) we know that:

$$\models \mathit{locEqual}(ld_1, ld_2) \quad \Leftrightarrow \quad ld_1 \equiv ld_2$$

and thus, by the congruence of $\equiv$, we know that for any expression $\alpha[ld]$ containing a location descriptor ld we can replace ld by an equal location descriptor ld':

$$\models \mathit{locEqual}(ld, ld') \quad \Rightarrow \quad \alpha[ld] \equiv \alpha[ld']$$

where $\alpha[\cdot] \in \mathit{Term}_\Sigma \cup \mathit{For}_\Sigma \cup \mathit{LocDesc}_\Sigma \cup \mathit{Upd}_\Sigma$.

Yet one note on simplification. The update simplifier should be applied in combination with basic simplifications of first order logic, to gain best results. For that reason, the following elementary simplifications will be applied silently for the rest of the thesis.

$\varphi \wedge \mathit{true}$	$\rightsquigarrow$	φ	$\varphi \wedge \neg\varphi$	$\rightsquigarrow$	false	$\neg\neg\varphi$	$\rightsquigarrow$	φ
$\varphi \wedge \mathit{false}$	$\rightsquigarrow$	false	$\varphi \wedge \varphi$	$\rightsquigarrow$	φ	$\forall x.\mathit{true}$	$\rightsquigarrow$	true
$\varphi \vee \mathit{true}$	$\rightsquigarrow$	true	$\varphi \vee \neg\varphi$	$\rightsquigarrow$	true	$\forall x.\mathit{false}$	$\rightsquigarrow$	false
$\varphi \vee \mathit{false}$	$\rightsquigarrow$	φ	$\varphi \vee \varphi$	$\rightsquigarrow$	φ	$\exists x.\mathit{true}$	$\rightsquigarrow$	true
$\mathit{true} \rightarrow \varphi$	$\rightsquigarrow$	φ	$\neg\mathit{true}$	$\rightsquigarrow$	false	$\exists x.\mathit{false}$	$\rightsquigarrow$	false
$\mathit{false} \rightarrow \varphi$	$\rightsquigarrow$	true	$\neg\mathit{false}$	$\rightsquigarrow$	true	$t \doteq t$	$\rightsquigarrow$	true

if true then $t1$ else $t2$	$\rightsquigarrow$	$t1$
if false then $t1$ else $t2$	$\rightsquigarrow$	$t2$

Example 1:

Let us start with a simple example: the update of a rigid function symbol (which is an observer function symbol with no dependencies, thus being constant in

each point across different states). The following axiom, specifying the rigidity of $obs \in FSym_{obs}$, is given:

$$\forall \bar{x}.locEqual(obs(\bar{x}), \texttt{empty}) \tag{4.1}$$

Updating obs in some point $\bar{t}$: $\{u\}obs(\bar{t})$ does not change the valuation of obs (as it is rigid). Hence, the update applies to the arguments only and the expression is equivalent to $obs(\{u\}\bar{t})$ (if obs was a nullary function symbol, we had the special case $\{u\}obs \equiv obs$). We will now see step by step, how the update simplifier enables us to come to this conclusion. If a rule is not applied to the whole expression but only a sub-expression, that sub-expression will be highlighted by a frame.

$$\begin{aligned}
&\{u\}obs(\bar{t}) \\
&\quad \overset{(\text{R2})}{\rightsquigarrow} \\
&(\{u\}obs)(\{u\}\bar{t}) \\
&\quad \overset{(\text{NR99})}{\rightsquigarrow} \\
&(\ \{\texttt{if}\ \neg \boxed{locDisjoint(dom(u), obs(\{u\}\bar{t}))}.u\}\ obs)(\{u\}\bar{t}) \\
&\quad \overset{(\text{NR131})}{\rightsquigarrow} \\
&(\ \{\texttt{if}\ \neg locSubset(dom(u) \cap obs(\{u\}\bar{t}), \texttt{empty}).u\}\ obs)(\{u\}\bar{t}) \\
&\quad \overset{(\text{R14})}{\rightsquigarrow} \\
&\text{if } \neg locSubset(dom(u) \cap \boxed{obs(\{u\}\bar{t})}, \texttt{empty}) \\
&\text{then } (\{u\}obs)(\{u\}\bar{t}) \\
&\text{else } obs(\{u\}\bar{t})
\end{aligned}$$

At this point we can apply the axiom (4.1) with $\bar{x}$ instantiated as $\{u\}\bar{t}$, giving $locEqual(obs(\{u\}\bar{t}), \texttt{empty})$. Accordingly, we can replace the framed $obs(\{u\}\bar{t})$ above by $\texttt{empty}$.

$$\begin{aligned}
&\text{if } \neg locSubset(\boxed{dom(u) \cap \texttt{empty}}, \texttt{empty}) \\
&\text{then } (\{u\}obs)(\{u\}\bar{t}) \\
&\text{else } obs(\{u\}\bar{t}) \\
&\quad \overset{(\text{NR138})}{\rightsquigarrow} \\
&\text{if } \neg \boxed{locSubset(\texttt{empty}, \texttt{empty})} \\
&\text{then } (\{u\}obs)(\{u\}\bar{t}) \\
&\text{else } obs(\{u\}\bar{t}) \\
&\quad \overset{(\text{NR109})}{\rightsquigarrow} \\
&\text{if } \boxed{\neg true} \\
&\text{then } (\{u\}obs)(\{u\}\bar{t}) \\
&\text{else } obs(\{u\}\bar{t}) \\
&\quad \overset{\text{FOL}}{\rightsquigarrow} \\
&\text{if } false \\
&\text{then } (\{u\}obs)(\{u\}\bar{t}) \\
&\text{else } obs(\{u\}\bar{t}) \\
&\quad \overset{\text{FOL}}{\rightsquigarrow} \\
&obs(\{u\}\bar{t})
\end{aligned}$$

We reached the expected expression. Of course, $\{u\}\bar{t}$ can probably still be simplified, depending on the specific $\bar{t}$ and u.

Example 2:

Recall the observer *nonNullArray*() of section 2.1 which checks the property of an array that it does not contain *null* entries. Clearly, for any array a, $nonNullArray(a)$ depends on the entries of a. This gives the following axiomatic definition of *nonNullArray*'s dependencies:

$$\forall a.locEqual(nonNullArray(a), \texttt{for}\ i.a[i]) \tag{4.2}$$

Let us add a note on arrays here: arrays are modelled using

- for each variable of type array a nullary location function symbol (constant) storing the reference to the array, e. g. $arr1 \in FSym_{loc} : T_1[\,]$
- the length operator on arrays: $len \in FSym_{loc} : T[\,] \to int$
- the array access operator: $[\,] \in FSym_{loc} : T[\,] \times int \to T$

For better readability, we write $a[i]$ instead of $[\,](a, i)$.

Continuing with the example, consider two arrays arr_1,$arr_2 \in FSym_{loc}$. We know that the two arrays are distinct:

$$\neg(arr_1 \doteq arr_2) \tag{4.3}$$

We want to know, whether changing arr_2 affects the *nonNullArray* property of arr_1:

$$\{arr_2[0] := null\}\ nonNullArray(arr_1)$$

This expression is reduced in 12 steps, including the application of the observer update rule (NR99), to:

$$\begin{array}{l}\text{if } locSubset(arr_2[0], \boxed{nonNullArray(arr_1)}) \\ \text{then } (\{arr_2[0] := null\}nonNullArray)(arr_1) \\ \text{else } nonNullArray(arr_1)\end{array}$$

Notice, that the framed $nonNullArray(arr_1)$ is a location descriptor, in contrast to the other occurrences of *nonNullArray*, which are terms. Accordingly, we can now apply the axiom (4.2). Instantiating the variable a with arr_1 we get: $locEqual(nonNullArray(arr_1), \texttt{for}\ i.arr_1[i])$ and thus, replacing $nonNullArray(arr_1)$ with $\texttt{for}\ i.arr_1[i]$ (framed in the following expression), we continue with

$$\begin{array}{l}\text{if } locSubset(arr_2[0], \boxed{\texttt{for}\ i.arr_1[i]}) \\ \text{then } (\{arr_2[0] := null\}nonNullArray)(arr_1) \\ \text{else } nonNullArray(arr_1)\end{array}$$

This will be rewritten in 3 steps to

$$\begin{array}{l}\text{if } \exists i.(i \doteq 0 \wedge \boxed{arr_1 \doteq arr_2}) \\ \text{then } (\{arr_2[0] := null\}nonNullArray)(arr_1) \\ \text{else } nonNullArray(arr_1)\end{array}$$

Now, we are almost done. Including the axiom (4.3), we can falsify $arr_1 \doteq arr_2$, and first order simplifications will transform

if $\exists i.(i \doteq 0 \wedge \boxed{false})$
then $(\{arr_2[0] := null\} nonNullArray)(arr_1)$
else $nonNullArray(arr_1)$

into

$nonNullArray(arr_1)$

Altogether, we could prove that the update $\{arr_2[0] := null\}$ does not affect the *nonNullArray* property of arr_1. We could finish the proof without having a definition for *nonNullArray*. All we used were its dependencies. In this way, the new update simplifier allows reasoning about abstract specifications.

Example 3:

This example will demonstrate the handling of anonymising updates. In the course of this, we will see, how a disjointness axiom can be used.

Consider the case of a class `List` which stores elements in some unspecified way. For the purpose of specification, the model field `elements` is introduced:

```
abstract class List
{
   //@ public model JMLValueSequence elements;

   ...
}
```

In KeY, the model field is translated to the unary observer function symbol *elements*, which takes as first argument the object (i. e. an instance of the class `List`) it belongs to. For example, if there is the variable `list1` of type `List`, then `list1` is translated to the nullary function symbol $list_1 \in FSym_{loc}$ and `list1.elements` to $elements(list_1)$ with $elements \in FSym_{obs}$. In fact, this is how ordinary attributes are translated, too – only that location function symbols are used for them.

Now, assume there is the local program variable `i` somewhere in the code. It is translated to $i \in FSym_{loc}$. This variable `i` is not part of the implementation of `elements`, which is expressed by the axiom

$$\forall l.locDisjoint(elements(l), i) \tag{4.4}$$

Arbitrary modifications to *elements* (expressed by the anonymising update $elements(list) := *1$) will therefore not affect i and consequently $i \doteq \{elements(list) := *1\}\, i$ is universally valid for any *list*. We are going to prove this property now.

$$\begin{aligned} & i \doteq \{elements(list) := *1\}\, i \\ & \qquad \rightsquigarrow^{+} \\ & i \doteq \text{if } locSubset(i, elements(list)) \\ & \quad \text{then } \{\texttt{everything} := *1\}\, i \\ & \quad \text{else } i \end{aligned}$$

At this point we can include the axiom (4.4). With l instantiated as *list* it reads: $locDisjoint(elements(list), i)$. Being always true, it can be conjoined

with any formula without changing the formula's truth value. That is why it can be included in the condition of the if-then-else term:

$$
\begin{array}{l}
i \doteq \text{if } (\mathit{locSubset}(i, \mathit{elements}(\mathit{list})) \wedge \boxed{\mathit{locDisjoint}(\mathit{elements}(\mathit{list}), i)}) \\
\quad \text{then } \{\texttt{everything} := *1\}\, i \\
\quad \text{else } i
\end{array}
$$

Let us now consider the simplification steps for the *locDisjoint* expression only:

$$
\begin{array}{l}
\mathit{locDisjoint}(\mathit{elements}(\mathit{list}), i) \\
\qquad \overset{\text{(NR131)}}{\rightsquigarrow} \\
\mathit{locSubset}(\boxed{\mathit{elements}(\mathit{list}) \cap i}, \texttt{empty}) \\
\qquad \overset{\text{(NR146)}}{\rightsquigarrow} \\
\mathit{locSubset}(\texttt{if } \mathit{locSubset}(i, \mathit{elements}(\mathit{list}))\,.\, i, \texttt{empty}) \\
\qquad \overset{\text{(NR111)}}{\rightsquigarrow} \\
\neg \mathit{locSubset}(i, \mathit{elements}(\mathit{list})) \vee \boxed{\mathit{locSubset}(i, \texttt{empty})} \\
\qquad \overset{\text{(NR120)}}{\rightsquigarrow} \\
\neg \mathit{locSubset}(i, \mathit{elements}(\mathit{list})) \vee \mathit{false} \\
\qquad \overset{\text{FOL}}{\rightsquigarrow} \\
\neg \mathit{locSubset}(i, \mathit{elements}(\mathit{list}))
\end{array}
$$

Remembering the whole expression again, we continue with:

$$
\begin{array}{l}
i \doteq \text{if } (\boxed{\mathit{locSubset}(i, \mathit{elements}(\mathit{list})) \wedge \neg \mathit{locSubset}(i, \mathit{elements}(\mathit{list}))}) \\
\quad \text{then } \{\texttt{everything} := *1\}\, i \\
\quad \text{else } i \\
\qquad \overset{\text{FOL}}{\rightsquigarrow} \\
i \doteq \text{if } \mathit{false} \\
\quad \text{then } \{\texttt{everything} := *1\}\, i \\
\quad \text{else } i \\
\qquad \overset{\text{FOL}}{\rightsquigarrow} \\
i \doteq i \\
\qquad \overset{\text{FOL}}{\rightsquigarrow} \\
\mathit{true}
\end{array}
$$

Hence, the proof of $i \doteq \{\mathit{elements}(\mathit{list}) := *1\}\, i$ could successfully be closed, using a disjointness axiom.

Chapter 5

Correctness

The rewrite rules given in chapter 3 are sound, in the sense of being equivalency transformations:

Proposition 5.1. *For every rewrite rule of the form*

$$lhs \quad \rightsquigarrow \quad rhs$$

given in chapter 3, the equivalence

$$lhs \quad \equiv \quad rhs$$

holds.

Recall, that $lhs \equiv rhs$ holds by definition if and only if for all Kripke structures $\mathcal{K}$, all states s in $\mathcal{K}$ and all variable assignments β it is true that $val_{\mathcal{K},s,\beta}(lhs) = val_{\mathcal{K},s,\beta}(rhs)$. Here, $val_{\mathcal{K},s,\beta}()$ stands for the valuation of terms, formulae, location descriptors or updates.

Proposition 5.1 has been proven by hand. Most of the proofs are rather trivial. For rule (NR100), for instance, the soundness, i. e. the equivalence:

$$(\{u\}f)(\bar{t}) \;\equiv\; f(\bar{t})$$

with $f \in FSym_{loc}$, follows directly from the semantics ($locVal_{\mathcal{K},s,\beta}((\{u\}f)(\bar{t})) := locVal_{\mathcal{K},s,\beta}(f(\bar{t}))$).

Similarly, the soundness of rule (NR137), i. e.:

$$\text{RESTRICT}(ld' := *n, ld) \;\equiv\; (ld' \cap ld) := *n$$

can be proven quite straightforwardly by applying the semantics of the valuation:

Proof. For all $(\mathcal{K}, s, \beta)$, ld, ld', n:

left hand side:

$$\begin{aligned}
& updVal_{\mathcal{K},s,\beta}(\text{RESTRICT}(ld' := *n, ld)) \\
= \;& \{\ (l,d) \in \underbrace{updVal_{\mathcal{K},s,\beta}(ld' := *n)}_{=\ \{\ (l,*(n)(l))\ \mid\ l \in locVal_{\mathcal{K},s,\beta}(ld')\ \}} \mid\ l \in locVal_{\mathcal{K},s,\beta}(ld)\ \} \\
= \;& \{\ (l,d) \in \{(l,*(n)(l)) \mid l \in locVal_{\mathcal{K},s,\beta}(ld')\} \mid\ l \in locVal_{\mathcal{K},s,\beta}(ld)\ \} \\
= \;& \{\ (l,*(n)(l)) \mid\ l \in locVal_{\mathcal{K},s,\beta}(ld') \wedge l \in locVal_{\mathcal{K},s,\beta}(ld)\ \}
\end{aligned}$$

right hand side:

$$
\begin{array}{ll}
 & updVal_{\mathcal{K},s,\beta}((ld' \cap ld) := *n) \\
= & \{\ (l, *(n)(l))\ \mid\ l \in locVal_{\mathcal{K},s,\beta}(ld' \cap ld)\ \} \\
= & \{\ (l, *(n)(l))\ \mid\ l \in locVal_{\mathcal{K},s,\beta}(ld') \cap locVal_{\mathcal{K},s,\beta}(ld)\ \} \\
= & \{\ (l, *(n)(l))\ \mid\ l \in locVal_{\mathcal{K},s,\beta}(ld')\ \wedge\ l \in locVal_{\mathcal{K},s,\beta}(ld)\ \}
\end{array}
$$

□

For some rules, we have to make sure clash resolution is preserved in the right way. One example is rule (NR133). We claim that:

$$\text{RESTRICT}(u_1 \| u_2, ld)\ \equiv\ \text{RESTRICT}(u_1, ld) \parallel \text{RESTRICT}(u_2, ld)$$

Proof. For all $(\mathcal{K}, s, \beta)$, u_1, u_2, ld:

left hand side:

$$
\begin{array}{ll}
 & updVal_{\mathcal{K},s,\beta}(\text{RESTRICT}(u_1 \| u_2, ld)) \\
= & \{\ (l,d) \in updVal_{\mathcal{K},s,\beta}(u_1 \| u_2)\ \mid\ l \in locVal_{\mathcal{K},s,\beta}(ld)\ \} \\
 & updVal_{\mathcal{K},s,\beta}(u_1 \| u_2) \\
= & \big(updVal_{\mathcal{K},s,\beta}(u_1) \cup updVal_{\mathcal{K},s,\beta}(u_2)\big) \\
 & \setminus \{\ (l,d) \in updVal_{\mathcal{K},s,\beta}(u_1)\ \mid\ (l,d') \in updVal_{\mathcal{K},s,\beta}(u_2) \text{ for some } d' \neq d\ \} \\
= & \{\ (l,d)\ \mid\ (l,d) \in updVal_{\mathcal{K},s,\beta}(u_1)\ \wedge\ (l,d') \notin updVal_{\mathcal{K},s,\beta}(u_2) \text{ for all } d' \neq d \\
 & \qquad \vee\ (l,d) \in updVal_{\mathcal{K},s,\beta}(u_2)\ \}
\end{array}
$$

thus:

$$
\begin{array}{ll}
 & updVal_{\mathcal{K},s,\beta}(\text{RESTRICT}(u_1 \| u_2, ld)) \\
= & \{\ (l,d)\ \mid\ l \in locVal_{\mathcal{K},s,\beta}(ld) \\
 & \qquad \wedge\ [(l,d) \in updVal_{\mathcal{K},s,\beta}(u_1)\ \wedge\ (l,d') \notin updVal_{\mathcal{K},s,\beta}(u_2) \text{ for all } d' \neq d \\
 & \qquad\quad \vee\ (l,d) \in updVal_{\mathcal{K},s,\beta}(u_2)]\ \}
\end{array}
$$

right hand side:

$$
\begin{array}{ll}
 & updVal_{\mathcal{K},s,\beta}(\text{RESTRICT}(u_1, ld) \parallel \text{RESTRICT}(u_2, ld)) \\
= & \big(\overbrace{updVal_{\mathcal{K},s,\beta}(\text{RESTRICT}(u_1, ld))}^{=:\ U_1} \cup \overbrace{updVal_{\mathcal{K},s,\beta}(\text{RESTRICT}(u_2, ld))}^{=:\ U_2}\big) \\
 & \setminus \{\ (l,d) \in U_1\ \mid\ (l,d') \in U_2 \text{ for some } d' \neq d\ \}
\end{array}
$$

now:

$U_1 = \{(l,d) \in updVal_{\mathcal{K},s,\beta}(u_1) \mid l \in locVal_{\mathcal{K},s,\beta}(ld)\}$

$U_2 = \{(l,d) \in updVal_{\mathcal{K},s,\beta}(u_2) \mid l \in locVal_{\mathcal{K},s,\beta}(ld)\}$

and thus:

$$
\begin{array}{ll}
 & updVal_{\mathcal{K},s,\beta}(\text{RESTRICT}(u_1, ld) \parallel \text{RESTRICT}(u_2, ld)) \\
= & \{\ (l,d)\ \mid\ [l \in locVal_{\mathcal{K},s,\beta}(ld) \wedge (l,d) \in updVal_{\mathcal{K},s,\beta}(u_1) \\
 & \qquad \wedge\ (l \notin locVal_{\mathcal{K},s,\beta}(ld) \vee (l,d') \notin updVal_{\mathcal{K},s,\beta}(u_2) \text{ for all } d' \neq d)] \\
 & \qquad \vee\ l \in locVal_{\mathcal{K},s,\beta}(ld) \wedge (l,d) \in updVal_{\mathcal{K},s,\beta}(u_2)\ \} \\
= & \{\ (l,d)\ \mid\ l \in locVal_{\mathcal{K},s,\beta}(ld) \\
 & \qquad \wedge\ [(l,d) \in updVal_{\mathcal{K},s,\beta}(u_1) \wedge\ (l,d') \notin updVal_{\mathcal{K},s,\beta}(u_2) \text{ for all } d' \neq d \\
 & \qquad\quad \vee\ (l,d) \in updVal_{\mathcal{K},s,\beta}(u_2)]\ \}
\end{array}
$$

□

Probably the most interesting case is the observer update rule (NR99). The soundness property for this rule reads:

$$(\{u\}g)(\bar{t})\ \equiv\ (\underbrace{\{\texttt{if}\ \neg locDisjoint(dom(u), g(\bar{t})).u\}}_{=:\ u'}g)(\bar{t})$$

where $g \in FSym_{obs}$ and $(\{u\}g)(\bar{t})$ is a term.

The proof makes use of the dependency of *observe* on *depends* as stated in the definition of Kripke structures (definition 2.6).

Proof. For all $(\mathcal{K}, s, \beta)$, u, g, $\bar{t}$:

left hand side:
$$\begin{array}{ll} & termVal_{\mathcal{K},s,\beta}((\{u\}g)(\bar{t})) \\ = & observe(s', g)(termVal_{\mathcal{K},s,\beta}(\bar{t})) \end{array}$$
where $s' = updVal_{\mathcal{K},s,\beta}(u)(s)$

right hand side:
$$\begin{array}{ll} & termVal_{\mathcal{K},s,\beta}((\{u'\}g)(\bar{t})) \\ = & observe(s'', g)(termVal_{\mathcal{K},s,\beta}(\bar{t})) \end{array}$$
where $s'' = updVal_{\mathcal{K},s,\beta}(u')(s)$

$$updVal_{\mathcal{K},s,\beta}(u') = \begin{cases} updVal_{\mathcal{K},s,\beta}(u) & , \ (\mathcal{K}, s, \beta) \models \neg locDisjoint(dom(u), g(\bar{t})) \\ \emptyset & , \ (\mathcal{K}, s, \beta) \not\models \neg locDisjoint(dom(u), g(\bar{t})) \end{cases}$$

case 1: $(\mathcal{K}, s, \beta) \models \neg locDisjoint(dom(u), g(\bar{t}))$

$$\begin{array}{ll} & updVal_{\mathcal{K},s,\beta}(u') = updVal_{\mathcal{K},s,\beta}(u) \\ \Rightarrow & s' = s'' \\ \Rightarrow & observe(s', g)(termVal_{\mathcal{K},s,\beta}(\bar{t})) = observe(s'', g)(termVal_{\mathcal{K},s,\beta}(\bar{t})) \\ \Rightarrow & (\{u\}g)(\bar{t}) \ \equiv \ (\{u'\}g)(\bar{t}) \qquad \square \end{array}$$

case 2: $(\mathcal{K}, s, \beta) \not\models \neg locDisjoint(dom(u), g(\bar{t}))$

$$\begin{array}{ll} & updVal_{\mathcal{K},s,\beta}(u') = \emptyset \\ \Rightarrow & s'' = s \end{array}$$

$$\begin{array}{ll} & (\mathcal{K}, s, \beta) \models locDisjoint(dom(u), g(\bar{t})) \\ \Rightarrow & locVal_{\mathcal{K},s,\beta}(dom(u)) \ \cap \ locVal_{\mathcal{K},s,\beta}(g(\bar{t})) \ = \ \emptyset \\ \Rightarrow & \{l \mid (l, d) \in updVal_{\mathcal{K},s,\beta}(u) \text{ for some } d \in \mathcal{D}\} \\ & \cap \ \underbrace{depends(s, g)(termVal_{\mathcal{K},s,\beta}(\bar{t}))}_{=: \ L} \ = \ \emptyset \end{array} \tag{5.1}$$

for all $l \in L$:
$$s'(l) = updVal_{\mathcal{K},s,\beta}(u)(s)(l) = \begin{cases} d & , \ \underbrace{(l, d) \in updVal_{\mathcal{K},s,\beta}(u)}_{\Leftrightarrow \ false \text{ by } (5.1)} \\ s(l) & , \text{ otherwise} \end{cases}$$
$$\Rightarrow \quad s'(l) = s(l)$$

thus, by the definitions of $\approx_L$ and *observe*:[1]
$$\begin{array}{ll} & s' \approx_L s = s'' \\ \Rightarrow & observe(s', g)(termVal_{\mathcal{K},s,\beta}(\bar{t})) = observe(s'', g)(termVal_{\mathcal{K},s,\beta}(\bar{t})) \\ \Rightarrow & (\{u\}g)(\bar{t}) \ \equiv \ (\{u'\}g)(\bar{t}) \qquad \square \end{array}$$

$\square$

[1] Another possible semantics, which had been considered but then rejected in favour of the current one, provided framing of location descriptors, too: for $s \approx_L s'$ or $s \approx_{L'} s'$ it forced $L = L'$, thus imposing the same restriction on *depends*() as on *observe*(). Location descriptors were framing themselves. In that semantics, rule (NR99) would be sound for location descriptors, as well. The proof would be the same, with *observe*() replaced by *depends*().

Chapter 6

Completeness

In the context of update simplification, the question of "completeness" is the question, whether all updates and all auxiliary (i. e. non-logical) constructors can be eliminated completely. These questions will be addressed in the present chapter.

6.1 Reduction to First Order Logic

We have seen, that there are different cases in which the update simplifier cannot get rid of updates or location descriptors entirely. These cases are, for updates:

- $(\{\texttt{everything} := *n\}f)(\bar{t}) \in \mathit{Term},\ f \in \mathit{FSym}_{loc}$
- $(\{u\}g)(\bar{t}) \in \mathit{Term},\ g \in \mathit{FSym}_{obs}$
- $(\{u\}g)(\bar{t}) \in \mathit{LocDesc},\ g \in \mathit{FSym}_{obs}$

and for location descriptors (apart from the appearance in anonymising updates):

- $\mathit{locSubset}(f(\bar{t}), G(\bar{s}))$, $f \in \mathit{FSym}_{loc}$, $G \in \mathit{FSym}_{obs} \cup \mathit{Alien}_{obs}$
- $\mathit{locSubset}(\texttt{everything}, ld)$
  where $ld \in \{\ G(\bar{t}),\ ld_1 \| ld_2,\ \texttt{for}\ x.ld'\ \}$
  with $G' \in \mathit{FSym}_{obs} \cup \mathit{Alien}_{obs}$
- $\mathit{locSubset}(G_1(\bar{t}_1) \cap \ldots \cap G_n(\bar{t}_n), ld)$, $n \geq 1$, $G_i \in \mathit{FSym}_{obs} \cup \mathit{Alien}_{obs}$
  where $ld \in \{\ f(\bar{t}),\ G'(\bar{s}),\ \texttt{empty},\ ld_1 \| ld_2,\ \texttt{for}\ x.ld'\ \}$
  with $f \in \mathit{FSym}_{loc}$, $G' \in \mathit{FSym}_{obs} \cup \mathit{Alien}_{obs}$

The auxiliary constructors *dom* and RESTRICT are eliminated entirely, substitutions are applied completely and updates remain in front of function symbols only.

Unfortunately, in the presence of observer symbols and anonymising updates, updates and location descriptors cannot be eliminated locally in general. This is a generic property of the logic and not related to the specific rewrite system.

Let us consider the different cases one by one. As we cannot refer to axioms locally, we assume an empty theory in the following.

$$(\{\texttt{everything} := *n\}f)(\bar{t}) \in \mathit{Term},\ f \in \mathit{FSym}_{loc}$$

For this expression, there is no equivalent term without updates. This is because such a term would basically be a FOL-expression (location descriptors make no difference here) and would therefore be valuated in a single state of a Kripke structure.

More formally: let a Kripke structure $\mathcal{K} = (\mathcal{D}, \prec, \mathcal{S}, \mathit{observe}, \mathit{depends}, *)$ for a signature with $f \in \mathit{FSym}_{loc}$ and, moreover, containing all symbols occurring in $\bar{t}$ be given. Assume $|\mathcal{D}| \geq 2$, which in turn implies $|\mathcal{S}| \geq 2$, as there must be a state for every possible interpretation of f as a function (see definition 2.6). Furthermore, assume there was a term T equivalent to $(\{\texttt{everything} := *n\}f)(\bar{t})$ but not containing any updates. Consider an arbitrary state $(\mathcal{K}, s, \beta)$. For simplicity, assume $\bar{t}$ does not contain updates.

Changing $\mathcal{K}$ to $\mathcal{K}'$ by changing $*$ such that

$$*_{\mathcal{K}'}(n)(f)(\mathit{termVal}_{\mathcal{K},s,\beta}(\bar{t})) \neq *_{\mathcal{K}}(n)(f)(\mathit{termVal}_{\mathcal{K},s,\beta}(\bar{t}))$$

does not change the valuation of T in s (formally, this means: $\mathit{termVal}_{\mathcal{K},s,\beta}(T) = \mathit{termVal}_{\mathcal{K}',s,\beta}(T)$), as neither s nor β changed and the valuation of FOL-terms depends on these only (see definition 2.13). In contrast to that, the valuation of $(\{\texttt{everything} := *n\}f)(\bar{t})$ *does* change. This is just how we chose $\mathcal{K}'$ ($\mathit{termVal}_{\mathcal{K}',s,\beta}((\{\texttt{everything} := *n\}f)(\bar{t}))$ being $*_{\mathcal{K}'}(n)(f)(\mathit{termVal}_{\mathcal{K}',s,\beta}(\bar{t}))$ and $\mathit{termVal}_{\mathcal{K}',s,\beta}(\bar{t}) = \mathit{termVal}_{\mathcal{K},s,\beta}(\bar{t})$). Hence, T is not equivalent to $(\{\texttt{everything} := *n\}f)(\bar{t})$.

Note, that the construction of $\mathcal{K}'$ is valid, as there are no restrictions on $*$ nor do other parts of a Kripke structure depend on it.

A similar argument applies to

$$(\{u\}g)(\bar{t}) \in \mathit{Term},\ g \in \mathit{FSym}_{obs}$$

assuming, the update is not empty ($\mathit{updVal}_{\mathcal{K},s,\beta}(u) \neq \emptyset$). Here, however, the value of the expression in the updated state *can* be linked to some other state due to the restrictions imposed on the *observe*() function (respect of the observer's dependencies defined by *depends*()). This is what the observer update rule (NR99) shown in section 3.3 takes advantage of. Taking the case, however, where an observer depends on all possible locations ($\mathit{depends}(s, g)(d) = \mathit{Locations}_{\mathcal{K}}$ for all states $s \in \mathcal{S}$ and all $d \in \mathcal{D}$ of a Kripke structure $\mathcal{K}$), the observer's value can change arbitrarily from state to state.

For observers in location descriptors

$$(\{u\}g)(\bar{t}) \in \mathit{LocDesc},\ g \in \mathit{FSym}_{obs}$$

we have the same situation as in terms with the observer depending on all locations. There are no restrictions for the *depends*() function. It follows that its value in any state of a Kripke structure can be changed arbitrarily (analogously to the $*$ function) without violating the definition of Kripke structures as used here.

It is the unrestrictedness of *depends*(), again, which makes it impossible to express formulae of the form

$$\mathit{locSubset}(f(\bar{t}), G(\bar{s})),\ f \in \mathit{FSym}_{loc},\ G \in \mathit{FSym}_{obs} \cup \mathit{Alien}_{obs}$$

or

$$locSubset(\texttt{everything}, G(\bar{t})),\ G' \in FSym_{obs} \cup Alien_{obs}$$

or

$$locSubset(G_1(\bar{t}_1) \cap \ldots \cap G_n(\bar{t}_n), ld),\ n \geq 1,\ G_i \in FSym_{obs} \cup Alien_{obs}$$
$$\text{where } ld \in \{\ f(\bar{t}),\ G'(\bar{s}),\ \texttt{empty},\ ld_1 \| ld_2,\ \texttt{for}\ x.ld'\ \}$$
$$\text{with } f \in FSym_{loc},\ G' \in FSym_{obs} \cup Alien_{obs}$$

without the use of – in this case – location descriptors, at least as long as the same observers do not appear on both sides of the subset relation ($locSubset(G(\bar{s}), G(\bar{s}))$ *can* of course be expressed without the use of location descriptors). The only way to refer to an observer's dependencies is by using location descriptors (as can be seen from the definition of the semantics). Changing *depends*() in an appropriate way would therefore affect the *locSubset* expressions above but not formulae without location descriptors (the formulae could even contain updates).

Actually, this is not *quite* true: unlike the $*$ function, the choice of *depends*() affects the set of valid *observe*() functions. This gives an indirect way to get some information about *depends*() using observers in terms. This dependency is only unidirectional, however. An *observe*() function which interprets all observer function symbols as constant functions is valid for all possible *depends*() functions. At least in Kripke structures with such an *observe*() function we can modify *depends*() freely without violating any constraints. Accordingly, the argument above can be applied.

As for

$$locSubset(\texttt{everything}, ld_1 \| ld_2)$$

$$locSubset(\texttt{everything}, \texttt{for}\ x.ld)$$

the same problem occurs on a higher level: thinking of reasoning about abstract specifications where we want to allow extensions to the signature, the valuation of `everything` would change. Even the enumeration of all function symbols would not help then (as *FSym* would not be fixed).

6.2 Updates of Observers

In the previous section, we have seen, that there are cases where we cannot get rid of updates completely. Now, let us focus on simplification of such updates.

In section 3.3 we defined a rule for simplifying non-recursive update applications on observer function symbols in terms. Let us recall the rule:

$$(\{u\}g)(\bar{t}) \quad \overset{1}{\rightsquigarrow} \quad (\ \{\texttt{if}\ \neg locDisjoint(dom(u), g(\bar{t})).u\}\ g)(\bar{t}) \qquad \text{(NR99)}$$
$$, g \in FSym_{obs}$$

This is, in fact, a sort of a cut rule. It makes use of the semantics of observers: if the values of the locations an observer depends on do not change from some given state to another, then the value of the observer is the same in both states, too. (This is formally defined in definition 2.6.)

Thus, if in an application of this rule in some state s the property $locDisjoint(dom(u), g(\bar{t}))$ holds, then the right hand side simplifies to $g(\bar{t})$. This is because, as the locations $g(\bar{t})$ depends on (described by the location descriptor $g(\bar{t})$) are not updated by u ($g(\bar{t})$ and $dom(u)$ are disjoint), the value of $g(\bar{t}^s)$ must be the same in the states s and $\{u\}s$ (where $\bar{t}^s$ denotes that $\bar{t}$ is valuated in s in both cases – the update applies to g only). A formal proof of this rule has been given in chapter 5.

Now, this rule is a bit of an all or nothing approach: either we drop the update u or we keep it as a whole. One might wonder whether it is possible to "simplify" u by chopping away irrelevant parts of it, if it cannot be omitted completely. For instance, having a term like $\{l_1 := r_1 \| l_2 := r_2\}o$ with $o \in FSym_{obs}$ and l_1,$l_2 \in FSym_{loc}$ where o does not depend on location l_2 it is tempting to simplify the term to $\{l_1 := r_1\}o$. Or, in general, one might want to restrict the update to the relevant part only by a rule like:

$$(\{u\}g)(\bar{t}) \quad \overset{1}{\rightsquigarrow} \quad (\ \{\text{RESTRICT}(u, g(\bar{t}))\}\ g)(\bar{t}) \qquad (\text{NR99}')$$
$$, g \in FSym_{obs}$$

where u is restricted to the set of locations $g(\bar{t})$ depends on using RESTRICT().

This rule, however, turns out to be unsound. The reason is that we examine $g(\bar{t})$'s dependencies in some state s but try to link its values in two other states, namely $\{u\}s$ and $\{\text{RESTRICT}(u, g(\bar{t}))\}s$. I. e. there are *three* states involved here. This is demonstrated best with an example.

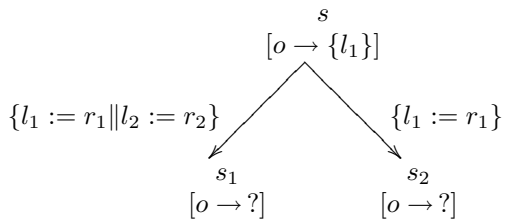


Figure 6.1: Dependencies of observer o in the states s, s_1 and s_2.

Reconsider the example above:

$$\{l_1 := r_1 \| l_2 := r_2\}o$$

with $o \in FSym_{obs}$ and l_1,$l_2 \in FSym_{loc}$. Consider the case of a state s such that:

$$s \models locEqual(o, l_1)$$

In particular, o does not depend on l_2 in s. This setting is visualised in figure 6.1. Let us assume that the value of l_1 is different from the value of r_1 in s and the same is true for l_2 and r_2, such that the values of the locations l_1 and l_2 are really changed by the update.

However, we do not know the dependencies of o in the updated states $s_1 = \{l_1 := r_1 \| l_2 := r_2\}s$ and $s_2 = \{l_1 := r_1\}s$. These two states differ exactly in the value of l_2. Now, it is possible, that o depends on l_2 in both s_1 and s_2. Then, as the value of l_2 would be different in these two states, the value of o might be different, too ($termVal_{s_1}(o) \neq termVal_{s_2}(o)$). In this case we

would have $termVal_s(\{l_1 := r_1 \| l_2 := r_2\}o) \neq termVal_s(\{l_1 := r_1\}o)$, rendering the "simplification" incorrect.

Note, that $updVal_s(\text{RESTRICT}(l_1 := r_1 \| l_2 := r_2,\, o)) = updVal_s(l_1 := r_1)$, thus this is a counterexample for rule (NR99′). For simplicity, the Kripke structure and the variable assignment are not named explicitly. They are of no interest here.

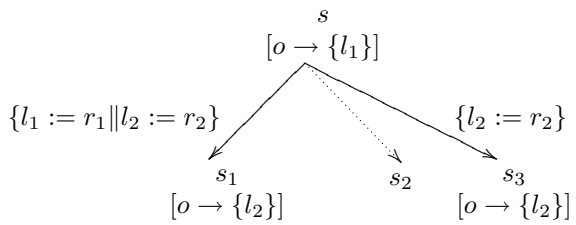


Figure 6.2: Fixed dependencies in s_1 and s_3.

The example can be driven even further: let us fix the dependencies of o in s_1 to be the singleton set $\{l_2\}$. Consider yet another state: $s_3 = \{l_2 := r_2\}s$ with the same dependencies for o (figure 6.2).

As s_1 and s_3 do not differ in l_2, the value of o has to be the same in both states: $termVal_{s_1}(o) = termVal_{s_3}(o)$.

Altogether, this gives the interesting case that, in this example, we have the state s with

$$s \models locEqual(o, l_1)$$

where

$$\begin{aligned} &\{l_1 := r_1 \| l_2 := r_2\}o \doteq \{l_2 := r_2\}o \\ &\text{but} \\ &\{l_1 := r_1 \| l_2 := r_2\}o \not\doteq \{l_1 := r_1\}o \end{aligned}$$

in s. That is, it is sound to drop the update of the location on which the observer depends but not the update of the location the observer does not depend on. Of course, this is just a special example and not the case in general.

Naturally, things change when we have *global* knowledge about an observer's dependencies. Having the *axiom*

$$\models locEqual(o, l_1)$$

we would know that (e. g. in figure 6.1) l_2 is not among o's dependencies in s_1 and s_2 and thus o has the same value in these two states.[1] In this case, we *could* simplify $\{l_1 := r_1 \| l_2 := r_2\}o$ to $\{l_1 := r_1\}o$. Yet, this involves global reasoning and is hence out of the scope of the update simplifier.

Considering the examples above carefully, one will notice that the whole problem with the rule (NR99′) arose from the missing link between the state in which we examined the observer's dependencies (s in the example above) and the states in which we valuated the observer (s_1 and s_2).

[1] Actually, $\models locDisjoint(o, l_2)$ would suffice.

In the original rule for observer updates, rule (NR99), we do not have this problem, as at most two states are involved: for a given $(\{u\}g)(\bar{t})$ the rewritten expression $(\{\texttt{if}\ \neg locDisjoint(dom(u), g(\bar{t})).u\}g)(\bar{t})$ is in any state s either equal to $(\{u\}g)(\bar{t})$ (where no change has taken place w. r. t. the original term) or to $g(\bar{t})$. In the latter case, the observer function symbol g is valuated in the state s itself, which is just the state in which we examine $g(\bar{t})$'s dependencies.

Following this example, we could try to fix rule (NR99′), rendering:

$$(\{u\}g)(\bar{t}) \quad \overset{1}{\rightsquigarrow} \quad (\ \{\text{RESTRICT}(u, \boxed{(\{u\}g)(\bar{t})})\}\ g)(\bar{t}) \qquad \text{(NR99}'')$$
$$, g \in FSym_{obs}$$

Notice the framed $(\{u\}g)(\bar{t})$ instead of $g(\bar{t})$ as in the previous rule (NR99′). Here, we examine g's dependencies in the updated state $\{u\}s$, which is the same state in which we valuate g on the left hand side of the rule. (The formal correctness proof is omitted here. It runs basically along the same lines as the case 2 of the proof of rule (NR99) given before.)

Yet, this rule has a major drawback: trying to simplify the *term* $(\{u\}g)(\bar{t})$ we introduced the *location descriptor* $(\{u\}g)(\bar{t})$. However, as there is no framing for location descriptors, we cannot say anything about $(\{u\}g)(\bar{t})$ unless we have axioms for it (which, typically, will not be the case). In this sense we make things even worse with this rule.

For these reasons we stick with rule (NR99) as the most useful one.

6.3 Update Equivalence

In this section we will have a look at update equivalence. It is often useful to decide, whether two updates are equivalent. Different equivalent updates in one expression can then be replaced by multiple occurrences of the same update. For example, $\{u_1\}obs \doteq \{u_2\}obs$ could be transformed to $\{u_1\}obs \doteq \{u_1\}obs$ if $u_1 \equiv u_2$. In this way, cases which are out of the local scope (and thus the scope of the update simplifier) could be handled. In the following, a sufficient (but not necessary) condition for the equivalence of updates will be given.

In [13], Philipp Rümmer defines a normal form for updates of the shape

$$\texttt{for}\ \bar{x}_1.\texttt{if}\ \varphi_1.f_1(\bar{t}_1) := r_1 \parallel \ \ldots\ \parallel \texttt{for}\ \bar{x}_k.\texttt{if}\ \varphi_k.f_k(\bar{t}_k) := r_k$$

where $\texttt{for}\ \bar{x}_i.u$ is a shorthand notation for $\texttt{for}\ x_{i,1}.\texttt{for}\ x_{i,2}.\cdots\texttt{for}\ x_{i,l_i}.u$. Rules for establishing this normal form are part of the update simplifier (mainly rules (R52), (R58), (R59), (R60) and (R64)). This normal form can be extended to cover anonymising updates, too. The result is

$$\texttt{for}\ \bar{x}_1.\texttt{if}\ \varphi_1.u_1 \parallel \ \ldots\ \parallel \texttt{for}\ \bar{x}_k.\texttt{if}\ \varphi_k.u_k \tag{6.1}$$

with

$$u_i = \begin{cases} f_i(\bar{t}_i) := r_i \\ \quad or \\ ld_i := *n_i \end{cases}$$

Unfortunately, the location descriptor in the anonymising update can be almost arbitrarily complex due to the intersection and difference of location descriptors. We will utilise this normal form for update comparison.

For the procedure which is presented in the following, another update constructor of [13] turns out to be useful: the REJECT constructor. In the new update simplifier it has been replaced by the RESTRICT constructor. As a matter of fact, RESTRICT$(u, \texttt{everything} \setminus dom(u'))$ would do just as well as REJECT(u, u'). However, for presentation purposes, the latter one is more suitable in the present context. That is why, to enable a clearer presentation of the following procedure, REJECT is introduced now.

The syntax of REJECT is

REJECT$(u, u') \in Upd_\Sigma$, for all u,$u' \in Upd_\Sigma$

The meaning of REJECT(u, u') is that part of update u which does not update locations also updated by u'. E. g. REJECT$(l_1 := a \| l_2 := b, l_1 := c)$ is equivalent to $l_2 := b$, and REJECT$(l_1 := a \| l_2 := b, l_3 := c)$ equivalent to $l_1 := a \| l_2 := b$, assuming l_1, l_2 and l_3 are all different locations. Note, that the assigned values do not need to differ, i. e. c could be equal to a and thus REJECT$(l_1 := a, l_1 := a)$ gives the empty update. The formal semantics is defined by

$$\begin{aligned} &updVal_{\mathcal{K},s,\beta}(\text{REJECT}(u, u')) \\ &\quad := \{(l, d) \in updVal_{\mathcal{K},s,\beta}(u) \mid (l, d') \notin updVal_{\mathcal{K},s,\beta}(u') \text{ for all } d' \in \mathcal{D}\} \end{aligned}$$

The following simplification rules can be defined for REJECT (taken from [13] and extended to anonymising updates):

$$\begin{aligned} \text{REJECT}(\texttt{skip}, u) &\rightsquigarrow \texttt{skip} \\ \text{REJECT}(f(\bar{t}) := r, u) &\rightsquigarrow \texttt{if } \neg locSubset(f(\bar{t}), dom(u)).(f(\bar{t}) := r) \\ \text{REJECT}(u_1 \| u_2, u) &\rightsquigarrow \text{REJECT}(u_1, u) \| \text{REJECT}(u_2, u) \\ \text{REJECT}(\texttt{if } \varphi.u', u) &\rightsquigarrow \texttt{if } \varphi.\text{REJECT}(u', u) \\ \text{REJECT}(\texttt{for } x.u', u) &\rightsquigarrow \texttt{for } x.\text{REJECT}(u', u) \\ & \quad , x \notin fv(u) \\ \text{REJECT}(ld := *n, u) &\rightsquigarrow (ld \setminus dom(u)) := *n \end{aligned}$$

These rules are equivalence transformations and eliminate REJECT completely (in combination with the rest of the update simplifier).

Another kind of transformation that will be needed is pulling conditions, quantifications and parallel constructors of anonymising updates into their location descriptors. All of these are equivalence transformations:

$$\begin{aligned} \texttt{if } \varphi.(ld := *n) &\rightsquigarrow (\texttt{if } \varphi.ld) := *n \\ \texttt{for } x.(ld := *n) &\rightsquigarrow (\texttt{for } x.ld) := *n \\ (ld_1 := *n) \| (ld_2 := *n) &\rightsquigarrow (ld_1 \| ld_2) := *n \end{aligned}$$

Now, a procedure will be defined to construct, for any two updates u_1 and u_2, a formula that gives a sufficient condition for the equivalence of the updates – i. e. whose universal validity implies $u_1 \equiv u_2$. The special case of the empty update `skip` will not be handled explicitly. It should be inserted, whenever any of the updates is empty (i. e. syntactically "nothing").

1. Establish normal form (6.1) for both u_1 and u_2. This is automatically done by the update simplifier. The result are the updates

$$\begin{aligned} u_1' &= \underbrace{\texttt{for}\ \bar{x}_{1,1}.\texttt{if}\ \varphi_{1,1}.u_{1,1}}_{=:\ \underline{u_{1,1}}} \parallel \ldots \parallel \underbrace{\texttt{for}\ \bar{x}_{1,k_1}.\texttt{if}\ \varphi_{1,k_1}.u_{1,k_1}}_{=:\ \underline{u_{1,k_1}}} \\ u_2' &= \underbrace{\texttt{for}\ \bar{x}_{2,1}.\texttt{if}\ \varphi_{2,1}.u_{2,1}}_{=:\ \underline{u_{2,1}}} \parallel \ldots \parallel \underbrace{\texttt{for}\ \bar{x}_{2,k_2}.\texttt{if}\ \varphi_{2,k_2}.u_{2,k_2}}_{=:\ \underline{u_{2,k_2}}} \end{aligned}$$

which are equivalent to u_1 and u_2 respectively.

$$u_{i,j} = \left\{ \begin{array}{l} f_{i,j}(\bar{t}_{i,j}) := r_{i,j} \\ \quad or \\ ld'_{i,j} := *n_{i,j} \end{array} \right.$$

2. Make the parallel sub-updates $\underline{u_{1,j}}$ of u_1' disjoint by simplifying

$$\begin{array}{l} \quad \text{REJECT}(\underline{u_{1,1}},\ \underline{u_{1,2}} \| \underline{u_{1,3}} \| \ldots \| \underline{u_{1,k_1}}) \\ \| \ \text{REJECT}(\underline{u_{1,2}},\ \underline{u_{1,3}} \| \ldots \| \underline{u_{1,k_1}}) \\ \| \ \ldots \\ \| \ \text{REJECT}(\underline{u_{1,k_1-1}},\ \underline{u_{1,k_1}}) \\ \| \ \underline{u_{1,k_1}} \end{array}$$

This is sound due to the *last win semantics* for parallel updates. Do the same for u_2'. The result are the updates u_1'' and u_2'' which are equivalent to u_1' and u_2' and thus to u_1 and u_2 respectively.

$$\begin{aligned} u_1'' &= \underbrace{\texttt{for}\ \bar{x}_{1,1}.\texttt{if}\ \psi_{1,1}.v_{1,1}}_{=:\ \underline{v_{1,1}}} \parallel \ldots \parallel \underbrace{\texttt{for}\ \bar{x}_{1,k_1}.\texttt{if}\ \psi_{1,k_1}.v_{1,k_1}}_{=:\ \underline{v_{1,k_1}}} \\ u_2'' &= \underbrace{\texttt{for}\ \bar{x}_{2,1}.\texttt{if}\ \psi_{2,1}.v_{2,1}}_{=:\ \underline{v_{2,1}}} \parallel \ldots \parallel \underbrace{\texttt{for}\ \bar{x}_{2,k_2}.\texttt{if}\ \psi_{2,k_2}.v_{2,k_2}}_{=:\ \underline{v_{2,k_2}}} \end{aligned}$$

with

$$v_{i,j} = \left\{ \begin{array}{l} f_{i,j}(\bar{t}_{i,j}) := r_{i,j} \\ \quad or \\ ld_{i,j} := *n_{i,j} \end{array} \right.$$

3. $u_1'' \equiv u_2''$ exactly if $u_1'' \subseteq u_2''$ and $u_2'' \subseteq u_1''$ which again is equivalent to

$$\underline{v_{1,1}} \subseteq u_2'' \wedge \ldots \wedge \underline{v_{1,k_1}} \subseteq u_2'' \quad \wedge \quad \underline{v_{2,1}} \subseteq u_1'' \wedge \ldots \wedge \underline{v_{2,k_2}} \subseteq u_1''$$

where $u \subseteq u'$ means $updVal_{\mathcal{K},s,\beta}(u) \subseteq updVal_{\mathcal{K},s,\beta}(u')$ for all $(\mathcal{K}, s, \beta)$.
Each $\underline{v_{1,i}} \subseteq u_2''$ (and analogously $\underline{v_{2,i}} \subseteq u_1''$) is implied by a formula which is built up in the following way.

   (a) In the case of $v_{1,i} = (f_{1,i}(\bar{t}_{1,i}) := r_{1,i})$
   [i. e. $\underline{v_{1,i}} = (\texttt{for}\ \bar{x}_{1,i}.\texttt{if}\ \psi_{1,i}.f_{1,i}(\bar{t}_{1,i}) := r_{1,i})$]:

$$\boxed{\forall \bar{y}. \Big[locSubset(f_{1,i}(\bar{y}), dom(\underline{v_{1,i}})) \\ \quad \rightarrow \Big(locSubset(f_{1,i}(\bar{y}), dom(u_2'')) \wedge \{\underline{v_{1,i}}\} f_{1,i}(\bar{y}) \doteq \{u_2''\} f_{1,i}(\bar{y}) \Big) \Big]}$$

with $|\bar{y}| = |\bar{t}_{1,i}|$ and $\bar{y} \cap (fv(\underline{v_{1,i}}) \cup fv(u_2'')) = \emptyset$.
Alternatively, with the left hand side of the implication simplified, one could say:

$$\forall \bar{y}. \Big[\Big(\exists \bar{x}_{1,i}. (\psi_{1,i} \wedge \bar{y} \doteq \bar{t}_{1,i}) \Big) \\ \rightarrow \Big(\mathit{locSubset}(f_{1,i}(\bar{y}), \mathit{dom}(u_2'')) \wedge \{\underline{v_{1,i}}\} f_{1,i}(\bar{y}) \doteq \{u_2''\} f_{1,i}(\bar{y}) \Big) \Big]$$

(b) In the case of $v_{1,i} = (ld_{1,i} := *n_{1,i})$
[i. e. $\underline{v_{1,i}} = (\texttt{for}\ \bar{x}_{1,i}.\texttt{if}\ \psi_{1,i}.(ld_{1,i} := *n_{1,i}))$]:

i. Drop all $\underline{v_{2,j}}$-s in u_2'' with $v_{2,j} = (ld_{2,j} := *n_{2,j})$ where $n_{2,j} \neq n_{1,i}$.

ii. Furthermore, drop all $\underline{v_{2,j}}$-s in u_2'' with $v_{2,j} = (f_{2,j}(\bar{t}_{2,j}) := r_{2,j})$.

iii. All remaining $\underline{v_{2,j}}$-s in the resulting u_2'' are of the shape $\texttt{for}\ \bar{x}_{2,j}.\texttt{if}\ \psi_{2,j}.(ld_{2,j} := *n_{1,i})$ (all with the same $n_{1,i}$ as in $\underline{v_{1,i}}$). Pull all `if`-, `for`- and $\|$-constructors into the location descriptor(s). The result is a single anonymising update of the form

$$\underbrace{(\texttt{for}\ \bar{x}_{2,l_1}.\texttt{if}\ \psi_{2,l_1}.ld_{2,l_1} \| \texttt{for}\ \bar{x}_{2,l_2}.\texttt{if}\ \psi_{2,l_2}.ld_{2,l_2} \| \ldots)}_{=:\ \widetilde{ld_2}} := *n_{1,i}$$

Now, $\underline{v_{1,i}} \subseteq u_2''$ is implied by

$$\mathit{locSubset}(\underbrace{\texttt{for}\ \bar{x}_{1,i}.\texttt{if}\ \psi_{1,i}.ld_{1,i}}_{=\ \mathit{dom}(\underline{v_{1,i}})}, \widetilde{ld_2})$$

The resulting formula can then be simplified in the usual way.

As mentioned before, the result is only a sufficient condition, but not a necessary one. The point, where equivalence is lost, is step 3b (or, more specifically, steps 3(b)i and 3(b)ii). Details can be found in the correctness proof for the procedure in the appendix.

A possible alternative for step 3(b)ii might be replacing all $\underline{v_{2,j}}$-s in u_2'' of the shape $\texttt{for}\ \bar{x}_{2,j}.\texttt{if}\ \psi_{2,j}.f_{2,j}(\bar{t}_{2,j}) := r_{2,j}$ with something like

$$\texttt{for}\ \bar{x}_{2,j}.\texttt{if}\ \psi_{2,j}.\texttt{if}\ c_1 \wedge c_2.(f_{2,j}(\bar{t}_{2,j}) := *n_{1,i})$$

where

$$c_1 \quad = \quad \bar{x}_{2,j} \doteq \min\ \bar{y}.\ \underbrace{\mathit{locSubset}(f_{2,j}(\bar{t}_{2,j}), \mathit{dom}((\texttt{if}\ \psi_{2,j}.f_{2,j}(\bar{t}_{2,j}) := r_{2,j})[\bar{y}/\bar{x}_{2,j}]))}_{\equiv\ \psi_{2,j}[\bar{y}/\bar{x}_{2,j}] \wedge \bar{t}_{2,j} \doteq (\bar{t}_{2,j}[\bar{y}/\bar{x}_{2,j}])}$$

$$c_2 \quad = \quad r_{2,j} \doteq (\{\texttt{everything} := *n_{1,i}\} f_{2,j})(\bar{t}_{2,j})$$

and, with $\bar{x} = (x_1, \ldots, x_n)$ and $\bar{y} = (y_1, \ldots, y_n)$, the expression $\alpha[\bar{y}/\bar{x}]$ stands for $\alpha[y_n/x_n] \cdots [y_1/x_1]$ and the expression $\bar{x} \doteq \min\ \bar{y}.\vartheta$ for $x_1 \doteq \min\ y_1.(x_2 \doteq \min\ y_2.(\cdots (x_n \doteq \min\ y_n.\vartheta)))$. This would turn step 3(b)ii into an equivalence transformation (while there would still be the non-equivalent step 3(b)i). The correctness of this alternative remains to be proved.

Three little examples will demonstrate now, what formulae are generated for different given updates.

Example 1:

u_1 : $l := a \parallel \texttt{if } \neg\varphi.l := b$
u_2 : $l := c$

This being a simple example, the updates already are in normal form (6.1) and step 1 can be skipped ($u_1 = u_1'$ and $u_2 = u_2'$).

To make the parallel sub-updates of u_1 disjoint (step 2), we have to simplify:

$$\text{REJECT}(l := a, \texttt{if } \neg\varphi.l := b) \parallel \texttt{if } \neg\varphi.l := b$$

The result is the update:

$$u_1'' : \quad \underbrace{\texttt{if } \varphi.l := a}_{\underline{v_{1,1}}} \parallel \underbrace{\texttt{if } \neg\varphi.l := b}_{\underline{v_{1,2}}}$$

For u_2, nothing has to be done and, hence, we have:

$$u_2'' : \quad \underbrace{l := c}_{\underline{v_{2,1}}}$$

We come to step 3. Recall, that

$$\begin{array}{ll} & u_1'' \equiv u_2'' \\ \Leftrightarrow & u_1'' \subseteq u_2'' \wedge u_2'' \subseteq u_1'' \\ \Leftrightarrow & \underline{v_{1,1}} \subseteq u_2'' \wedge \underline{v_{1,2}} \subseteq u_2'' \wedge \underline{v_{2,1}} \subseteq u_1'' \end{array}$$

Let us build up the formulae for $\underline{v_{1,1}} \subseteq u_2''$, $\underline{v_{1,2}} \subseteq u_2''$ and $\underline{v_{2,1}} \subseteq u_1''$ (step 3a in each case).

- $\underline{v_{1,1}} \subseteq u_2''$:

$$\boxed{\begin{array}{l} \mathit{locSubset}(l, \mathit{dom}(\texttt{if } \varphi.l := a)) \\ \quad \rightarrow \Big(\mathit{locSubset}(l, \mathit{dom}(l := c)) \ \wedge \ \{\texttt{if } \varphi.l := a\}\, l \doteq \{l := c\}\, l\Big) \end{array}}$$

  This formula is simplified by the update simplifier to:

$$\varphi \rightarrow (\text{if } \varphi \text{ then } a \text{ else } l) \doteq c$$

- $\underline{v_{1,2}} \subseteq u_2''$:

$$\boxed{\begin{array}{l} \mathit{locSubset}(l, \mathit{dom}(\texttt{if } \neg\varphi.l := b)) \\ \quad \rightarrow \Big(\mathit{locSubset}(l, \mathit{dom}(l := c)) \ \wedge \ \{\texttt{if } \neg\varphi.l := b\}\, l \doteq \{l := c\}\, l\Big) \end{array}}$$

  This formula is simplified by the update simplifier to:

$$\neg\varphi \rightarrow (\text{if } \neg\varphi \text{ then } b \text{ else } l) \doteq c$$

- $\underline{v_{2,1}} \subseteq u_1''$:

$$\boxed{\begin{array}{l} \mathit{locSubset}(l, \mathit{dom}(l := c)) \\ \quad \rightarrow \Big(\mathit{locSubset}(l, \mathit{dom}(\texttt{if } \varphi.l := a \parallel \texttt{if } \neg\varphi.l := b)) \\ \qquad \wedge \ \{l := c\}\, l \doteq \{\texttt{if } \varphi.l := a \parallel \texttt{if } \neg\varphi.l := b\}\, l\Big) \end{array}}$$

  This formula is simplified by the update simplifier to:

$$c \doteq \text{if } \neg\varphi \text{ then } (\text{if } \neg\varphi \text{ then } b \text{ else } l) \text{ else } (\text{if } \varphi \text{ then } a \text{ else } l)$$

Hence, the (simplified) condition for the equivalence of u_1 and u_2 is:

$$\begin{aligned} & [\varphi \rightarrow (\text{if } \varphi \text{ then } a \text{ else } l) \doteq c] \\ \wedge\ & [\neg\varphi \rightarrow (\text{if } \neg\varphi \text{ then } b \text{ else } l) \doteq c] \\ \wedge\ & [c \doteq \text{if } \neg\varphi \text{ then (if } \neg\varphi \text{ then } b \text{ else } l) \text{ else (if } \varphi \text{ then } a \text{ else } l)] \end{aligned}$$

Some further first order reasoning and reasoning about terms yields:

$$(\varphi \rightarrow a \doteq c) \wedge (\neg\varphi \rightarrow b \doteq c) \wedge (c \doteq \text{if } \neg\varphi \text{ then } b \text{ else } a)$$

or simply:

$$(\varphi \rightarrow a \doteq c) \wedge (\neg\varphi \rightarrow b \doteq c)$$

This makes perfectly sense, as u_1 assigns a to l if φ is true, and if φ is false, b is assigned to l (remember, that with parallel composition, later assignments override earlier ones). On the other hand, u_2 always assigns c to the same location l.

Example 2:

Let o_1,o_2,$o_3 \in FSym_{obs}$.

$$\begin{aligned} u_1 :&\quad o_1 := *1 \parallel o_2 := *2 \\ u_2 :&\quad o_3 := *2 \end{aligned}$$

The simplified condition in this case is

$$locSubset(o_1, o_2) \wedge locSubset(o_2, o_3) \wedge locSubset(o_3, o_2)$$

One can see, that this formula indeed implies the equivalence of u_1 and u_2: if, in u_1, the effects of $o_1 := *1$ are overridden by $o_2 := *2$ (which is the case if all locations in o_1 are also in o_2, i. e. $locSubset(o_1, o_2)$) then u_1 is equivalent to $o_2 := *2$ and u_1 and u_2 are equivalent if o_2 and o_3 are equal ($locSubset(o_2, o_3) \wedge locSubset(o_3, o_2)$).

However, one can also see, that this condition is not a necessary one. The two updates can be equivalent, even if the formula is not universally valid. This could, for instance, be the case, if the domain of all Kripke structures for the considered signature and theory had exactly one element (which could be forced by an axiom like $\forall x, y.x \doteq y$). In that case, an anonymising update of the form $ld := *n$ would have the same effects for any n. Consequently, $o_1 := *1$ would be equivalent to $o_3 := *2$ if o_1 and o_3 were equal and thus u_1 and u_2 would be equivalent if o_1 and o_3 were equal and o_2, say, empty.

Example 3:

$$\begin{aligned} u_1 :&\quad \texttt{for } x.f(x) := g(x) \\ u_2 :&\quad \texttt{for } x.\texttt{if } \varphi.f(x) := g(x) \end{aligned}$$

The formula φ may contain free occurrences of x. The result of the procedure here is the somewhat lengthy formula:

$$
\begin{array}{l}
\forall y.\big[\ (\exists x.[y \doteq x \wedge \varphi]) \\
\quad \rightarrow \\
\quad \big(\ (\exists x.y \doteq x) \\
\quad\quad \wedge \\
\quad\quad (\ \text{if } \varphi[(\min\, x.(y \doteq x \wedge \varphi))/x] \\
\quad\quad\quad \text{then if } y \doteq \min\, x.(y \doteq x \wedge \varphi) \text{ then } g(\min\, x.(y \doteq x \wedge \varphi)) \text{ else } f(y) \\
\quad\quad\quad \text{else } f(y) \\
\quad\quad) \doteq (\text{if } y \doteq \min\, x.(y \doteq x) \text{ then } g(\min\, x.y \doteq x) \text{ else } f(y)) \\
\quad \big) \\
\big] \\
\wedge \\
\forall y.\big[\ \exists x.(y \doteq x) \\
\quad \rightarrow \\
\quad \big(\ \exists x.(y \doteq x \wedge \varphi) \\
\quad\quad \wedge \\
\quad\quad (\text{if } y \doteq \min\, x.(y \doteq x) \text{ then } g(\min\, x.y \doteq x) \text{ else } f(y)) \\
\quad\quad \doteq \\
\quad\quad (\ \text{if } \varphi[(\min\, x.(y \doteq x \wedge \varphi))/x] \\
\quad\quad\quad \text{then if } y \doteq \min\, x.(y \doteq x \wedge \varphi) \text{ then } g(\min\, x.(y \doteq x \wedge \varphi)) \text{ else } f(y) \\
\quad\quad\quad \text{else } f(y)\ ) \\
\quad \big) \\
\big]
\end{array}
$$

First order reasoning and reasoning about terms finally gives the following condition:

$$\forall x.\varphi$$

which is just what one would expect.

Deciding the equivalence of two updates in *every* case (i. e. a sufficient and necessary condition) would enable us to eliminate updates in yet another way, using an approach similar to Skolemisation where equivalent expressions are replaced by the same Skolem-like symbol. Of course, the resulting expressions would not be equivalent but, in the case of formulae, equisatisfiable. For example, an expression like $\{\texttt{everything} := *n\}f$ with $f \in FSym_{loc}$ could be replaced everywhere by a fresh rigid symbol (i. e. an observer symbol with an empty set of dependencies, making it constant in each point), say f^{*n}. This has already been proposed in an early version of [14]. To extend this approach to observers – i. e. replacing $\{u\}g$ where $g \in FSym_{obs}$ with something like g^u – we would need to be able to decide the equivalence of updates. The reason is that $\{u_1\}g$ and $\{u_2\}g$ denote the same thing if $u_1 \equiv u_2$ and would accordingly have to be replaced by the same Skolem-like symbol g^{u_1}.

One has to be careful with axioms, however. Consider the following example: let $g \in FSym_{obs}$ and $f \in FSym_{loc}$ be given. Assume, there is the expression $\{u\}g$ which we want to replace and the axiom

$$\forall x.(g(x) \doteq f(x))$$

We have to transfer the information given hereby for g to the new symbol g^u. To this end, we apply the update u to the axiom:

$$\{u\}\forall x.(g(x) \doteq f(x))$$

pull the update inside:

$$\forall x.((\{u\}g)(x) \doteq \{u\}f(x))$$

and Skolemise $\{u\}g$ here, too:

$$\forall x.(g^u(x) \doteq \{u\}f(x))$$

The result is then added as a new axiom (still retaining the old one). Note, that the u in $\{u\}f(x)$ can hopefully be eliminated.

Let, for example, u be the update $f(5) := 1$. Then the value of $(\{f(5) := 1\}g)(5)$ is 1 and

$$(\{f(5) := 1\}g)(5) \doteq 2$$

is unsatisfiable. On the other hand, for an arbitrary rigid symbol g^u the term $g^u(5)$ can have any value and

$$g^u(5) \doteq 2$$

is satisfiable. This is fixed by adding the updated axiom which has in this case the shape:

$$\forall x.(g^u(x) \doteq (\text{if } x \doteq 5 \text{ then } 1 \text{ else } f(x)))$$

Now, $g^u(x)$ is equal to $(\{f(5) := 1\}g)(x)$ for all x.

All of this needs to be worked out in detail, however, to see all the pitfalls and to find out, whether this approach is really feasible at all.

Chapter 7

Related Work

The main problem which we are tackling with the specification of dependencies, namely the question which actions do or do not affect certain locations, is known as the *frame problem* [5]. It has been commonly approached by formulating *frame axioms* which specify, what parts of the involved logic (e. g. locations in our case) do not change. In the appropriate setting, such frame axioms can even be generated systematically [15]. Having started in artificial intelligence, the frame problem was examined in the context of program specifications, too [16]. In [17] the authors used dynamic logic to implement different solutions to the frame problem. All these approaches focused on the frame problem itself. Data abstraction and modularity were not among the considered aspects.

To handle the frame problem in a modular context, Leino introduced *abstract variables* – functions depending on other functions (or program variables) – and a *depends clause* for stating their dependencies explicitly [6]. In our context they correspond to observer function symbols and dependency axioms. However, Leino approached the problem from the viewpoint of a specification language and did not establish a logical framework with a respective calculus. Continuing his idea, he developed the notion of *data groups* [7] which represent "sets of variables" and can be used in specifications. This corresponds to location sets and observer location descriptors in our setting. Again, the focus is on the specification language. Data groups are used in JML [18, 11]. JML's *model fields* (a version of Leino's abstract variables) correspond to our observer function symbols, data groups to observer location descriptors and the `in` and `maps-into` clauses specify dependencies like *locSubset* statements.

Another solution to the frame problem using dependencies in a modal logic similar to a propositional dynamic logic is proposed in [19]. The modalities are annotated with generic "actions", literals (propositional atoms or their negations) can explicitly be specified to be dependent on actions. A tableau calculus which makes use of the dependencies is presented. The definition of the dependence relation $\rightsquigarrow$ (definition 6 in [19]) is similar to the semantics of *depends* and *observe* (definition 2.6 in this thesis). As the dependent symbols are literals, they are, unlike observers, always parameterless. Also, the calculus of [19] works on the meta-level, while the update simplifier is located on the syntactic level.

Dynamic dependencies, i. e. dependencies where the set of locations an *abstract variable* depends on can be different in different (program) states, are used in [20]. Our observers are dynamic in this sense, too. The dynamic de-

pendencies of [20], however, have a specific form, which allows "modifies list desugaring", i. e. the 'static' unfolding of dependencies.

A more generic approach for varying dependencies, are *dynamic frames* [8]. Dynamic frames correspond to observer location descriptors, but do not carry parameters. Analogously, the meaning of dynamic frames corresponds to location sets. Disjointness of frames can be specified explicitly, which is what we do with the *locDisjoint* predicate. *Regional logic* is mainly based on the idea of dynamic frames (called *regions* here), too [21]. Dynamic frames were implemented, e. g. in Spec# [22] where framing axioms are automatically generated from explicit framing information [23]. Another implementation is described in [24]. Here, dependencies are not specified explicitly, but are deduced from so-called *accessibility predicates*, which specify that a location may be read from or written to.

The idea to encode *observers* [4, Section 3.1], i. e. dependent symbols carrying parameters, in logic using "new uninterpreted function(s)" (which is what we do with our location function symbols) can be found, e. g. in [25]. It has been implemented in the ESC/Java2 program verification tool [26].

Postponed frame axiom generation as described in [27], where frame axioms are generated when a query occurs, has some distant resemblance to our approach of putting respective (in-)dependency requirements (expressed via *locSubset*) into conditions of conditional updates, location descriptors or if-then-else terms. In contrast to our approach, this is done on the meta-level in [27] and the dependencies are implicit (all symbols occurring in the query).

There are also approaches, where effects on dependent symbols do not happen immediately along with changes to their dependencies, but have to be triggered explicitly with *pack* statements [28]. The underlying idea is not to see *model fields* as abstract functions (which is what we do with observers), but rather as stored values, which have to be updated separately.

Update simplification in KeY has been introduced in [13], however without observers and explicit dependencies. Explicit dependencies in the context of KeY have been examined in [29]. In contrast to using *locEqual* statements in axioms, dependencies are attached to symbols directly, i. e. syntactically, which reduces modularity compared to the possibility of under-specification using *locSubset* or *locDisjoint*. [29] also presents two simplification rules for update application on dependent symbols (i. e. observer update simplification rules). Unlike in the case of the update simplifier, these rules do not operate locally. The semantics for observers and dependencies as used in this thesis has been introduced in an internal document of the Institute for Theoretical Computer Science, University of Karlsruhe [14].

Chapter 8

Conclusions

8.1 Summary

Within this thesis an extended update simplifier has been presented, which can handle observer symbols, location descriptors and anonymising updates. It consists of some 190 rules (the original version, given in [13], has approximately 80 rules, most of which have been incorporated in this new version). Its correctness has been proven (and some exemplary proofs have been included in this thesis).

The update simplifier reduces given expressions to expressions of first order logic with dependency conditions (*locSubset* statements in conditional terms, updates and location descriptors) in most cases. It has been shown, however, that there are theoretical limits to the update simplifier's power: not all updates can be eliminated in the presence of observer symbols. The pitfalls with simplification of updates which are applied to observer function symbols in terms ($(\{u\}g)(\bar{t}) \in Term$) have been demonstrated and it has been shown, why there cannot be simplification rules for non-recursive update application on observer location descriptors ($(\{u\}g)(\bar{t}) \in LocDesc$).

In the course of work for this thesis, the rule set has been implemented in Maude [30, 31], a programming language and framework which allows an easy implementation of term rewriting systems and comes along with an interpreter. Various examples have been explored with this implementation, some of which have been presented here. In this way, it has been shown that the update simplifier successfully supports the process of abstract reasoning by reducing updated expressions to expressions using dependency conditions, which have then to be resolved at a higher level (often involving global reasoning).

8.2 Future Work

First of all, the update simplifier needs to be implemented in KeY. Additionally, appropriate mechanisms for handling the remaining observer updates and for dependency resolution have to be developed (e. g. appropriate sequent calculus rules). These have to be combined with the update simplifier in a reasonable way.

Framing for location descriptors should be considered. This could lead to further possibilities of simplification. However, it has to be evaluated, whether

the resulting restrictions are acceptable for the purposes of program verification in KeY.

Another interesting aspect which should be looked into more deeply, is the determination of update equivalence and the connected questions of feasibility and usefulness of Skolemisation.

Appendix

In this appendix, the correctness of the procedure in section 6.3, which gives a sufficient condition for the equivalence of updates, is proved.

To simplify notation in the following proof, some additional notation will be used. Recall, that semantic updates are consistent in the sense that they contain, for any location l, at most one tuple (l, d) (definition 2.10). Hence, for any Kripke structure $\mathcal{K}$ with domain $\mathcal{D}$, any semantic update U can be regarded as a partial function

$$U : \mathit{Locations}_{\mathcal{K}} \nrightarrow \mathcal{D}$$

where

$$U(l) = \begin{cases} d & , \ (l,d) \in U \\ \text{undefined} & , \ \text{otherwise} \end{cases} \qquad \text{for all } l \in \mathit{Locations}_{\mathcal{K}}$$

With this interpretation, $\mathrm{dom}(U)$ gives the domain of the semantic update U. It is easy to see, that

$$\mathit{updVal}_{\mathcal{K},s,\beta}(\mathit{dom}(u)) = \mathrm{dom}(\mathit{updVal}_{\mathcal{K},s,\beta}(u))$$

for any update u.

Proposition A.1. *The procedure for determining a sufficient condition for the equivalence of updates as given in section 6.3 is sound, i. e. for any two updates u_1 and u_2 let the resulting condition be ψ_{u_1,u_2}, then:*

$$\big(\models \psi_{u_1,u_2}\big) \ \Rightarrow \ \big(u_1 \equiv u_2\big)$$

Proof. As $\equiv$ is an equivalence relation, it is enough to show the equivalence of updates u_1'' and u_2'' which are equivalent to u_1 and u_2 respectively.

Step 1 of the procedure:

The proof that the update simplifier establishes the normal form (6.1) is skipped here. The equivalences $u_1' \equiv u_1$ and $u_2' \equiv u_2$ follow from the soundness of the update simplification rules.

Step 2:

Equivalency (R54) in [13] states that

$$u \| u' \equiv \mathrm{REJECT}(u, u') \| u'$$

Repeated application of this equivalency yields $u_1'' \equiv u_1'$ and $u_2'' \equiv u_2'$. Notice, that $\underline{v_{i,j}}$ is the result of the simplification of $\mathrm{REJECT}(\underline{u_{i,j}}, \dots)$. Consequently, all $\underline{v_{1,j}}$-s are disjoint (equally, all $\underline{v_{2,j}}$-s).

Step 3:

Quite obviously, by the semantics of $u \| u'$:

$$\mathrm{dom}(updVal_{\mathcal{K},s,\beta}(u \| u')) = \mathrm{dom}(updVal_{\mathcal{K},s,\beta}(u)) \cup \mathrm{dom}(updVal_{\mathcal{K},s,\beta}(u'))$$

By the semantics of REJECT we know that:

$$(l,d) \in updVal_{\mathcal{K},s,\beta}(\mathrm{REJECT}(u,u')) \Rightarrow l \notin \mathrm{dom}(updVal_{\mathcal{K},s,\beta}(u'))$$

Hence, by the construction of u_1'':

$$(l,d) \in updVal_{\mathcal{K},s,\beta}(\underline{v_{1,i}}) \Rightarrow l \notin \mathrm{dom}(updVal_{\mathcal{K},s,\beta}(\underline{v_{1,j}})) \quad \text{for all } j \neq i$$

and:

$$updVal_{\mathcal{K},s,\beta}(u_1'') = updVal_{\mathcal{K},s,\beta}(\underline{v_{1,1}}) \cup \ldots \cup updVal_{\mathcal{K},s,\beta}(\underline{v_{1,k_1}})$$

which means, that each location is updated by at most one $\underline{v_{1,i}}$ and consequently there are no clashes between the different $\underline{v_{1,i}}$-s. The same applies to u_2''.

It follows that:

$$\begin{array}{ll}
 & updVal_{\mathcal{K},s,\beta}(u_1'') \subseteq updVal_{\mathcal{K},s,\beta}(u_2'') \quad \text{for all } (\mathcal{K},s,\beta) \\
\Leftrightarrow & \left[updVal_{\mathcal{K},s,\beta}(\underline{v_{1,1}}) \cup \ldots \cup updVal_{\mathcal{K},s,\beta}(\underline{v_{1,k_1}})\right] \subseteq updVal_{\mathcal{K},s,\beta}(u_2'') \\
 & \text{for all } (\mathcal{K},s,\beta) \\
\Leftrightarrow & updVal_{\mathcal{K},s,\beta}(\underline{v_{1,1}}) \subseteq updVal_{\mathcal{K},s,\beta}(u_2'') \ \wedge \ \ldots \\
 & \wedge\ updVal_{\mathcal{K},s,\beta}(\underline{v_{1,k_1}}) \subseteq updVal_{\mathcal{K},s,\beta}(u_2'') \quad \text{for all } (\mathcal{K},s,\beta) \\
\Leftrightarrow & \quad updVal_{\mathcal{K},s,\beta}(\underline{v_{1,1}}) \subseteq updVal_{\mathcal{K},s,\beta}(u_2'') \quad \text{for all } (\mathcal{K},s,\beta) \\
 & \text{and } updVal_{\mathcal{K},s,\beta}(\underline{v_{1,2}}) \subseteq updVal_{\mathcal{K},s,\beta}(u_2'') \quad \text{for all } (\mathcal{K},s,\beta) \\
 & \vdots \\
 & \text{and } updVal_{\mathcal{K},s,\beta}(\underline{v_{1,k_1}}) \subseteq updVal_{\mathcal{K},s,\beta}(u_2'') \quad \text{for all } (\mathcal{K},s,\beta)
\end{array}$$

Hence:

$$u_1'' \equiv u_2''$$
$$\Leftrightarrow$$
$$\underline{v_{1,1}} \subseteq u_2'' \text{ and } \ldots \text{ and } \underline{v_{1,k_1}} \subseteq u_2'' \quad \text{and} \quad \underline{v_{2,1}} \subseteq u_1'' \text{ and } \ldots \text{ and } \underline{v_{2,k_2}} \subseteq u_1''$$

with $u \subseteq u'$ meaning $updVal_{\mathcal{K},s,\beta}(u) \subseteq updVal_{\mathcal{K},s,\beta}(u')$ for all $(\mathcal{K},s,\beta)$, i. e. each $u \subseteq u'$ being universally valid.

Step 3a [$\underline{v_{1,i}} = (\texttt{for } \bar{x}_{1,i} \texttt{.if } \psi_{1,i}.f_{1,i}(\bar{t}_{1,i}) := r_{1,i})$]:

Here, we have the special case that:

$$(l,e) \in updVal_{\mathcal{K},s,\beta}(\underline{v_{1,i}}) \ \Rightarrow \ l = (f_{1,i}, \bar{d}) \text{ for some } \bar{d} \in \mathcal{D}^{\alpha(f_{1,i})}$$

Therefore, we can conclude:

$$
\begin{array}{ll}
 & updVal_{\mathcal{K},s,\beta}(\underline{v_{1,i}}) \subseteq updVal_{\mathcal{K},s,\beta}(u_2'') \quad \text{for all } (\mathcal{K},s,\beta) \\
\Leftrightarrow & \text{for all } (l,e) : (\overline{l},e) \in updVal_{\mathcal{K},s,\beta}(\underline{v_{1,i}}) \Rightarrow (l,e) \in updVal_{\mathcal{K},s,\beta}(u_2'') \\
 & \text{for all } (\mathcal{K},s,\beta) \\
\Leftrightarrow & \text{for all } \bar{d}, e : ((f_{1,i},\bar{d}),e) \in updVal_{\mathcal{K},s,\beta}(\underline{v_{1,i}}) \Rightarrow ((f_{1,i},\bar{d}),e) \in updVal_{\mathcal{K},s,\beta}(u_2'') \\
 & \text{for all } (\mathcal{K},s,\beta) \\
\Leftrightarrow & \text{for all } \bar{d} : (f_{1,i},\bar{d}) \in \mathrm{dom}(updVal_{\mathcal{K},s,\beta}(\underline{v_{1,i}})) \\
 & \qquad \Rightarrow \Big((f_{1,i},\bar{d}) \in \mathrm{dom}(updVal_{\mathcal{K},s,\beta}(u_2'')) \\
 & \qquad\quad \text{and } updVal_{\mathcal{K},s,\beta}(\underline{v_{1,i}})((f_{1,i},\bar{d})) = updVal_{\mathcal{K},s,\beta}(u_2'')((f_{1,i},\bar{d})) \Big) \\
 & \text{for all } (\mathcal{K},s,\beta) \\
\Leftrightarrow & \models \forall \bar{y}. \Big[locSubset(f_{1,i}(\bar{y}), dom(\underline{v_{1,i}})) \\
 & \qquad \rightarrow \Big(locSubset(f_{1,i}(\bar{y}), dom(u_2'')) \wedge \{\underline{v_{1,i}}\} f_{1,i}(\bar{y}) \doteq \{u_2''\} f_{1,i}(\bar{y}) \Big) \Big]
\end{array}
$$

with $|\bar{y}| = |\bar{d}|$ and $\bar{y} \cap (fv(\underline{v_{1,i}}) \cup fv(u_2'')) = \emptyset$. Notice, that

$$updVal_{\mathcal{K},s,\beta}(\{u\}f(\bar{y})) = updVal_{\mathcal{K},s,\beta}(u)((f, termVal_{\mathcal{K},s,\beta}(\bar{y})))$$

for $(f, termVal_{\mathcal{K},s,\beta}(\bar{y})) \in \mathrm{dom}(updVal_{\mathcal{K},s,\beta}(u))$ and that $\{u\}f(\bar{y}) \equiv (\{u\}f)(\bar{y})$.

Step 3b:

With the disjointness of the $\underline{v_{2,i}}$-s and the resulting equality:

$$updVal_{\mathcal{K},s,\beta}(u_2'') = updVal_{\mathcal{K},s,\beta}(\underline{v_{2,1}}) \cup \ldots \cup updVal_{\mathcal{K},s,\beta}(\underline{v_{2,k_2}})$$

we know that, with $\{l_1, l_2, \ldots\} \subseteq \{1, 2, \ldots, k_2\}$:

$$updVal_{\mathcal{K},s,\beta}(\underline{v_{2,l_1}} \| \underline{v_{2,l_2}} \| \ldots) \subseteq updVal_{\mathcal{K},s,\beta}(u_2'')$$

and, hence:

$$
\begin{array}{ll}
 & updVal_{\mathcal{K},s,\beta}(u) \subseteq updVal_{\mathcal{K},s,\beta}(\underline{v_{2,l_1}} \| \underline{v_{2,l_2}} \| \ldots) \quad \text{for all } (\mathcal{K},s,\beta) \\
\Rightarrow & updVal_{\mathcal{K},s,\beta}(u) \subseteq updVal_{\mathcal{K},s,\beta}(u_2'') \quad \text{for all } (\mathcal{K},s,\beta)
\end{array}
$$

That means that we can drop arbitrary $\underline{v_{2,i}}$-s from u_2'' without loosing the sufficiency of the resulting condition. This justifies steps 3(b)i and 3(b)ii. Further notes concerning these two steps are given after this proof.

Step 3(b)iii:

By the semantics of anonymising updates and *locSubset*:

$$
\begin{array}{ll}
 & updVal_{\mathcal{K},s,\beta}(ld := *n) \subseteq updVal_{\mathcal{K},s,\beta}(ld' := *n) \quad \text{for all } (\mathcal{K},s,\beta) \\
\Leftrightarrow & \{(l, *(n)(l)) \mid l \in locVal_{\mathcal{K},s,\beta}(ld)\} \subseteq \{(l, *(n)(l)) \mid l \in locVal_{\mathcal{K},s,\beta}(ld')\} \\
 & \text{for all } (\mathcal{K},s,\beta) \\
\Leftrightarrow & locVal_{\mathcal{K},s,\beta}(ld) \subseteq locVal_{\mathcal{K},s,\beta}(ld') \quad \text{for all } (\mathcal{K},s,\beta) \\
\Leftrightarrow & \models locSubset(ld, ld')
\end{array}
$$

That is why, together with the correctness of the rules for pulling `if-`, `for-` and ‖-constructors into location descriptors, step 3(b)iii is sound.

Finally, as

$$\models \varphi \text{ and } \models \psi \text{ if and only if } \models \varphi \wedge \psi$$

we can conjoin the partial formulae generated according to steps 3a and 3b and show the universal validity of the resulting formula. □

Now, let us say a few words about steps 3(b)i and 3(b)ii.

Step 3(b)ii is not an equivalence transformation, as can be seen by the counter-example:

$$f(\bar{t}) := (\{\texttt{everything} := *n\}f)(\bar{t}) \quad \equiv \quad f(\bar{t}) := *n$$

with $f \in FSym_{loc}$, which shows a case, where an elementary update is equivalent to an anonymising one.

A counter-example for the equivalence of step 3(b)ii has been shown in Example 2 of section 6.3: the case where all Kripke structures have a domain with only one element. Let us consider this step 3(b)ii a little closer. Given certain assumptions, it turns out to be an equivalence transformation, after all.

We want to show:

$$\begin{array}{ll} & updVal_{\mathcal{K},s,\beta}(ld := *n) \subseteq updVal_{\mathcal{K},s,\beta}(u \| ld' := *m) \quad \text{for all } (\mathcal{K}, s, \beta) \\ \Rightarrow & updVal_{\mathcal{K},s,\beta}(ld := *n) \subseteq updVal_{\mathcal{K},s,\beta}(u) \quad \text{for all } (\mathcal{K}, s, \beta) \end{array}$$

if $n \neq m$ and $\mathrm{dom}(updVal_{\mathcal{K},s,\beta}(u)) \cap locVal_{\mathcal{K},s,\beta}(ld') = \emptyset$.

Assumption 1: Assume, that there is no $ld'' := *m$ in ld, nor in ld', nor in u.

Assumption 2: Furthermore, assume that if

$$updVal_{\mathcal{K},s,\beta}(ld := *n) \subseteq updVal_{\mathcal{K},s,\beta}(u \| ld' := *m)$$

but

$$updVal_{\mathcal{K},s,\beta}(ld := *n) \not\subseteq updVal_{\mathcal{K},s,\beta}(u)$$

for some Kripke structure $\mathcal{K}$ and some s and β, then there is a Kripke structure $\mathcal{K}'$ with $|\mathcal{D}| > 1$ where this is the case, too, i. e.:

$$updVal_{\mathcal{K}',s',\beta'}(ld := *n) \subseteq updVal_{\mathcal{K}',s',\beta'}(u \| ld' := *m)$$

but

$$updVal_{\mathcal{K}',s',\beta'}(ld := *n) \not\subseteq updVal_{\mathcal{K}',s',\beta'}(u)$$

for some s' and β'.

Under these assumptions, the implication can be proved.

Proposition A.2. *Given, assumption 1 and assumption 2 hold, the following implication is true:*

$$\begin{array}{ll} & updVal_{\mathcal{K},s,\beta}(ld := *n) \subseteq updVal_{\mathcal{K},s,\beta}(u \| ld' := *m) \quad \text{for all } (\mathcal{K}, s, \beta) \\ \Rightarrow & updVal_{\mathcal{K},s,\beta}(ld := *n) \subseteq updVal_{\mathcal{K},s,\beta}(u) \quad \text{for all } (\mathcal{K}, s, \beta) \end{array}$$

if $n \neq m$ *and* $\mathrm{dom}(updVal_{\mathcal{K},s,\beta}(u)) \cap locVal_{\mathcal{K},s,\beta}(ld') = \emptyset$.

Proof. Assume, the left hand side holds, i. e.:

$$updVal_{\mathcal{K},s,\beta}(ld := *n) \subseteq updVal_{\mathcal{K},s,\beta}(u \| ld' := *m) \quad \text{for all } (\mathcal{K}, s, \beta) \qquad (1)$$

Assume further, that the right hand side does not hold, i. e.:

$$(l, d) \in updVal_{\mathcal{K},s,\beta}(ld := *n) \text{ but } (l, d) \notin updVal_{\mathcal{K},s,\beta}(u) \qquad (2)$$

for some (l, d) and $(\mathcal{K}, s, \beta)$. Then, by (1):

$$(l, d) \in updVal_{\mathcal{K},s,\beta}(ld' := *m)$$

i. e.: $l \in locVal_{\mathcal{K},s,\beta}(ld')$ and $*_{\mathcal{K}}(m)(l) = d$. Based on assumption 2, we can assume, without loss of generality, that $|\mathcal{D}| > 1$ for $\mathcal{D}$ of $\mathcal{K}$. Consider the Kripke structure $\mathcal{K}'$ which is equal to $\mathcal{K}$ except for $*_{\mathcal{K}'}(m)(l) = d' \neq d$. As, by assumption 1, there is no $ld'' := *m$ in ld or ld' or u, their interpretations do not change, i. e.:

$$locVal_{\mathcal{K},s,\beta}(ld) = locVal_{\mathcal{K}',s,\beta}(ld)$$
$$locVal_{\mathcal{K},s,\beta}(ld') = locVal_{\mathcal{K}',s,\beta}(ld')$$
$$updVal_{\mathcal{K},s,\beta}(u) = updVal_{\mathcal{K}',s,\beta}(u)$$

and thus: $l \in locVal_{\mathcal{K}',s,\beta}(ld')$. Accordingly, we have:

$$(l, d') \in updVal_{\mathcal{K}',s,\beta}(ld' := *m)$$

which implies:

$$(l, d') \in updVal_{\mathcal{K}',s,\beta}(u \| ld' := *m)$$

and hence, with $d \neq d'$ and the consistency of semantic updates:

$$(l, d) \notin updVal_{\mathcal{K}',s,\beta}(u \| ld' := *m)$$

However, with (2) and $locVal_{\mathcal{K},s,\beta}(ld) = locVal_{\mathcal{K}',s,\beta}(ld)$, we know that:

$$(l, d) \in updVal_{\mathcal{K}',s,\beta}(ld := *n)$$

which is a contradiction to (1). That means that assumption (2) was wrong and the proposition true under the given assumptions. □

Let us, finally, have a look at what can go wrong if the assumptions do not hold.

Example 1: Assumption 1 does not hold, as u contains a $ld'' := *m$:
Let $\varphi = (\{l := *n\}l \doteq \{l := *m\}l)$ and $l \in FSym_{loc}$. Then we have:

$$updVal_{\mathcal{K},s,\beta}(l := *n) \subseteq updVal_{\mathcal{K},s,\beta}(\texttt{if } \neg\varphi.(l := *n) \| (\texttt{if } \varphi.l) := *m)$$
for all $(\mathcal{K}, s, \beta)$

but:

$$updVal_{\mathcal{K},s,\beta}(l := *n) \not\subseteq updVal_{\mathcal{K},s,\beta}(\texttt{if } \neg\varphi.(l := *n))$$
for $\mathcal{K}$ with $*(n)(l) = *(m)(l)$

and thus, the implication of proposition A.2 does not hold.

Example 2: Assumption 2 does not hold, the implication does not hold only in certain Kripke structures with $|\mathcal{D}| = 1$:
Let $\varphi = (\forall x,y\,.\, x \doteq y)$ and $l \in FSym_{loc}$. Then we have:

$$updVal_{\mathcal{K},s,\beta}(l := *n) \subseteq updVal_{\mathcal{K},s,\beta}(\texttt{if } \neg\varphi.(l := *n) \| (\texttt{if } \varphi.l) := *m)$$
for all $(\mathcal{K}, s, \beta)$

because in Kripke structures with $|\mathcal{D}| = 1$ (in which case only φ is true – and then in all states) $l := *n$ has the same effect as $l := *m$. On the other hand:

$$updVal_{\mathcal{K},s,\beta}(l := *n) \not\subseteq updVal_{\mathcal{K},s,\beta}(\texttt{if } \neg\varphi.(l := *n))$$
for $\mathcal{K}$ with $|\mathcal{D}| = 1$

Hence, here too, the implication of proposition A.2 does not hold.

Bibliography

[1] Robert W. Floyd. Assigning meanings to programs. *Proc. Symposia in Applied Mathematics*, 19:19–32, 1967.

[2] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, oct 1969.

[3] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, aug 1975.

[4] Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Asp. Comput.*, 19(2):159–189, 2007.

[5] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Melzter and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, 1969.

[6] K. Rustan M. Leino. Toward reliable modular programs. PhD thesis Caltech-CS-TR-95-03, California Institute of Technology, Pasadena, California, 1995.

[7] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In Craig Chambers, editor, *Proceedings of the 13th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, volume 33, pages 144–153. ACM Press, 1998.

[8] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2006. ISBN: 3-540-37215-6.

[9] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer, 2007. ISBN: 3-540-68977-X.

[10] Homepage of the KeY-project: `http://key-project.org/`.

[11] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, and Werner Dietl. *JML Reference Manual (DRAFT)*, jul 2008. Available on `http://www.jmlspecs.org/`.

[12] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. Foundations of Computing. MIT Press, October 2000. ISBN: 0-262-08289-6.

[13] Philipp Rümmer. Sequential, parallel, and quantified updates of first-order structures; in: Proving and disproving in dynamic logic for Java. Licentiate Thesis 2006–26L, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, 2006. ISSN: 1652-876X.

[14] Bernhard Beckert, Richard Bubel, Christian Engel, Peter H. Schmitt, Mattias Ulbrich, and Benjamin Weiß. Classification of symbols. Internal document, Institute for Theoretical Computer Science, Department of Computer Science, University of Karlsruhe, 2008.

[15] Edwin P. D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning (KR'89)*, pages 324–332. Morgan Kaufmann, 1989.

[16] Alex Borgida, John Mylopoulos, and Raymond Reiter. On the frame problem in procedure specifications. *IEEE Trans. Software Eng.*, 21(10):785–798, 1995.

[17] Dongmo Zhang and Norman Foo. Frame problem in dynamic logic. *Journal of Applied Non-Classical Logics*, 15(2):215–239, 2005.

[18] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006. Also: Iowa State University, Department of Computer Science, TR #98-06-rev29, January 2006.

[19] Marcos A. Castilho, Olivier Gasquet, and Andreas Herzig. Formalizing action and change in modal logic i: the frame problem. *J. Log. Comput.*, 9(5):701–735, 1999.

[20] K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, 2002.

[21] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In Jan Vitek, editor, *ECOOP*, volume 5142 of *Lecture Notes in Computer Science*, pages 387–411. Springer, 2008. ISBN: 978-3-540-70591-8.

[22] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Barthe et al. [32], pages 49–69. ISBN: 3-540-24287-2.

[23] Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. An automatic verifier for java-like programs based on dynamic frames. In José Luiz Fiadeiro and Paola Inverardi, editors, *FASE*, volume 4961 of *Lecture Notes in Computer Science*, pages 261–275. Springer, 2008. ISBN: 978-3-540-78742-6.

[24] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. In *Proceedings, 10th Workshop on Formal Techniques for Java-like Programs (FTfJP 2008)*, pages 1–12, 2008.

[25] David R. Cok. Reasoning with specifications containing method calls and model fields. *Journal of Object Technology*, 4(8):77–103, 2005.

[26] David R. Cok and Joseph Kiniry. ESC/Java2: Uniting ESC/Java and JML. In Barthe et al. [32], pages 108–128. ISBN: 3-540-24287-2.

[27] Dongmo Zhang and Norman Foo. Interpolation properties of action logic: Lazy-formalization to the frame problem. In Sergio Flesca, Sergio Greco, Nicola Leone, and Giovambattista Ianni, editors, *JELIA*, volume 2424 of *Lecture Notes in Computer Science*, pages 357–368. Springer, 2002. ISBN: 3-540-44190-5.

[28] K. Rustan M. Leino and Peter Müller. A verification methodology for model fields. In Peter Sestoft, editor, *ESOP*, volume 3924 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2006. ISBN: 3-540-33095-X.

[29] Richard Bubel, Reiner Hähnle, and Peter H. Schmitt. Specification predicates with explicit dependency information. In Bernhard Beckert and Gerwin Klein, editors, *VERIFY*, volume 372 of *CEUR Workshop Proceedings*, pages 28–43. CEUR-WS.org, 2008.

[30] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude – A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007. ISBN: 978-3-540-71940-3.

[31] Homepage of the Maude project: `http://maude.cs.uiuc.edu/`.

[32] Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors. *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop, CASSIS 2004, Marseille, France, March 10-14, 2004, Revised Selected Papers*, volume 3362 of *Lecture Notes in Computer Science*. Springer, 2005. ISBN: 3-540-24287-2.