Internal Report 2011–11 December 2011

Universiteit Leiden
Opleiding Informatica

Automated Verification

of Programs with Pointers

Jurriaan Rot

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Automated Verification of Programs with Pointers

Jurriaan Rot

Master thesis
LIACS – Leiden University

Supervisors:
prof. dr. F.S. de Boer
dr. M.M. Bonsangue

Abstract

We present a fully automated method for the verification of annotated recursive programs with dynamic pointer structures. Assertions are expressed in a dialect of dynamic logic extended with nominals and tailored to heap structures, in which one can express complex properties such as reachability. Verification conditions are generated using a novel calculus for computing the strongest post-condition of statements manipulating the heap, such as dynamic allocation and field-assignment. Further, we introduce a new decidable tableaux-based method to automatically check these verification conditions.

Contents

Chapter 1

Introduction

Dynamic allocation and assignments result in complicated structures in the heap, which may easily lead to subtle bugs. Even in small programs unexpected behaviour may arise through notorious events such as aliasing. Programs with complicated heap structures are ubiquitous; think for example of object-oriented languages supporting the dynamic creation of objects, which may refer to each other through instance fields. Unfortunately but not unexpectedly, it is in general very hard to formally reason about the heap in these kinds of programs.

In this thesis we nevertheless approach this problem and consider the verification of annotated pointer programs, which are simply sets of method declarations complemented with *contracts*, pre- and postcondition specifications in an assertion language. Partial correctness of such programs with respect to their outline means that whenever on the execution of a method the precondition is satisfied initially, then after the execution, if it terminates – hence *partial* correctness – the postcondition of the method holds. In order to focus on the main aspects, we introduce a simple imperative programming language which supports assigments including dynamic allocation, and standard constructs as while loops and recursive procedure calls.

As assertion languages are usually based on first-order logic, *automated* verification of annotated programs is mostly undecidable. In this thesis we introduce a fully automated formal method for checking the correctness of annotated pointer programs. Our starting point is the development of an expressive yet *decidable* logic, based on propositional dynamic logic [8]. Heap structures can be regarded as labelled transition systems, where states are labeled with variables and transitions are labeled with fields. This is reflected in our logic, where program variables are modeled as a specific kind

of propositional variables and the basic modalities are fields. In our setting propositional variables thus are *nominals*, i.e., true in precisely one location of the heap – reminiscent of hybrid logic [4] – and transitions (fields) are deterministic. To deal with these specific features, we introduce a tableaux method for deciding entailment of formulas.

We then introduce syntactic characterizations of the strongest postconditions of assignments of variables and fields, and of dynamic allocation. This basic calculus is extended to the abstract execution of programs to automatically generate verification conditions, which can be proven or disproven by a specifically tailored decision method for entailment of formulas. Our method is fully tool-supported with a working prototype written for the rewrite engine Maude [7].

Related work There are many logics and proof systems for verifying properties of the heap, of which separation logic [12] is an increasingly popular one. Further proving correctness of object-oriented programs has been studied intensively (see e.g. [2, 1]). However these logics are in general undecidable; for example, in [5] it is shown that even the purely propositional part of separation logic is not decidable. Further, often these logics are based on first-order logic and thus require additional constructs to express reachability. Consequently the correctness of pointer programs still defies automation.

A decidable logic similar to ours was introduced in [3], also including nominals and deterministic fields. While basic reachability is expressible in this logic, there are no nested modalities, in contrast to the logic introduced in this thesis. Further [3] contains no characterization of weakest preconditions or strongest postconditions.

In the recent work [14] a hybrid μ-calculus is used to express strongest pre- and postconditions of low level pointer updates. Preconditions are however expressed in an undecidable dialect including converse modalities. The strongest postconditions are expressed in a decidable hybrid μ-calculus (c.f. [13]); however this is a very complex logic including global modalities. Further in contrast to [14] we focus on a high-level programming language. Finally [14] lacks a proof of correctness of the strongest postconditions.

As for the decidability result of our dynamic logic, there are many existing tools for satisfiability of propositional dynamic logic [9], including methods for (hybrid) logics with nominals [10, 13]. Unfortunately none of these results fit to the exact characteristics of our logic, which, apart from nominals, includes deterministic fields. We developed a tableaux method

which exploits precisely these characteristics resulting in a modular approach, which first resolves propositional connectives, then resolves nominals and finally checks eventualities.

Summary of the contributions This thesis is based on the paper *F.S. de Boer, M.M. Bonsangue, J. Rot. Automated Verification of Recursive Pointer Programs.* Our main contribution is a tool-supported[1] method for automated verification of pointer programs, consisting of the following core ingredients:

- An expressive program logic for heap abstraction
- A syntactic characterization of the strongest postconditions of assignments together with a rigorous proof of correctness
- A tableaux method for deciding entailment

Outline This thesis is structured as follows. In Chapter 2 we introduce a formalization of heap structures and define the programming language. In Chapter 3 we introduce and discuss our logic and define the notion of annotated programs. Chapter 4 details the tableaux method for semantic entailment of formulas. Then in Chapter 5 we introduce the strongest postcondition calculus and show how to apply this to the verification of annotated programs, concluding with a report of checking a simple program on our prototype implementation. Finally in Chapter 6 we conclude with various directions for future research.

[1]The full source code can be downloaded from http://www.liacs.nl/ jrot/verify

Chapter 2

Heaps, programs

In the first section of this chapter we formalize the notion of a heap in terms of transition systems, where edges are labelled with fields and states with variables. Then the second section defines our programming language, together with a formal operational semantics.

2.1 Heaps

Throughout this thesis we assume an infinite set V of *variables* including a distinguished variable $nil \in V$ and ranged over by x, y, z, and an infinite set of fields F ranged over by f, g. Further we have finite sets $V_P \subset V$ and $F_P \subset F$ of *program variables* and *program fields*, respectively. A *heap* H is a pair $\langle v, h \rangle$ of a variable assignment $v : V \to \mathbb{N}$ and a field assignment $f : F \to (\mathbb{N} \to \mathbb{N})$. We write $H(x)$ for $v(x)$, and $H(f)$ for $h(f)$. For a set of variables Var and a set of fields Fld we denote by $\mathcal{R}_H(Var, Fld)$ the set of *reachable states* in H starting from these variables over fields in Fld, defined as the least set X such that $H(Var) \subseteq X$ and $H(f)(X) \subseteq X$ for all $f \in Fld$. For technical convenience we assume that for every heap H, $\mathcal{R}_H(V, F)$ is finite. We denote variable update by $H[x := n]$, global field update and store update by $H[f := \rho]$ and $\rho[n := m]$, respectively, where $\rho : \mathbb{N} \to \mathbb{N}$, and, finally, a local field update by $H[f := H(f)[n := m]]$. We use the standard notation and definition of simultaneous assignments and updates.

A brief discussion is in order. Heaps, as introduced above, have both an infinite domain and an infinite range. The motivation for having an infinite number of variables and fields is to simplify the introduction of logical variables in the definition of strongest postconditions, described below. The

reason we chose for natural numbers to model "locations" is for an easy implementation of dynamic allocation. Variable and field mappings are total functions; however, the requirement that there is a variable $nil \in V$ allows to distinguish between variables (or fields) which are null, and variables which are not.

2.2 A simple programming language

In this section we introduce a simple imperative programming language which supports recursion, assignment of variables and update of fields, and dynamic allocation. In order to proceed we assume finite sets $V_P \subset V$ and $F_P \subset F$ of *program variables* and *program fields* respectively. A *recursive program* P is a collection

$$P = \{p_1 :: S_1, \ldots, p_n :: S_n\}$$

of procedure declarations where each S_i is a statement defined by the following grammar:

$$\begin{aligned} S &::= \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi} \mid \text{ while } B \text{ do } S \text{ od} \mid p \mid S_1; S_2 \mid \epsilon \\ A &::= x := y \mid x := y.f \mid x.f := y \mid x := \text{new} \\ B &::= x = y \mid x \neq y \end{aligned}$$

where x, y are program variables ranging over V_P, f ranges over F_P and p is a procedure identifier ranging over the procedure names of P. The expressions A are called *assignments*; note that these include dynamic allocation. Let us first discuss informally the semantics of this language. The statements $x := y$ and $x := y.f$ are basic assignments, updating x to point to (the location referred to by) y or $y.f$ respectively. Field update $x.f := y$ affects the *location* to which x points, and may thus also affect aliases of x. Note that we have defined *nil* as a variable whereas it should be constant, and so we disallow field updates to variables aliased with *nil*. Dynamic allocation $x :=$ new assigns to x a new location unreachable in the heap from any other program variables over program fields. The meaning of conditional statements, sequential composition and while loops is as expected. A procedure identifier p is simply a call to the procedure p. Finally ϵ is the empty statement, which we include for technical convenience.

At first sight the navigation expressions in the basic assignments and conditions in this language may seem to be rather limited. More general expressions and updates, however, can be encoded – in a sequential setting

– using only basic assignments and composition. For example, a statement $x := y.f_{i_1} \dots f_{i_k}$ is encoded as $x := y.f_{i_1}; x := x.f_{i_2}; x := x.f_{i_3}; \dots; x := x.f_{i_k}$. In fact, even the operation $x := y$ is not strictly necessary in presence of the others, as it can be encoded as $z.f := y; x := z.f$.

We turn to the formal operational semantics of this language, described by means of a transition relation $\rightarrow$ between configurations which are pairs $\langle H, S\rangle$ of a heap H and a statement S. The semantics of assignments is described by the following transitions:

$$\langle H, x := y\rangle \rightarrow \langle H[x := H(y)], \epsilon\rangle$$

$$\langle H, x := y.f\rangle \rightarrow \langle H[x := H(f)(H(y))], \epsilon\rangle$$

Field update $x.f := y$ affects aliases of x. As mentioned above we require that x is not equal to *nil* for a correct execution of such an update:

$$\frac{H(x) \neq H(nil)}{\langle H, x.f := y\rangle \rightarrow \langle H[f := H(f)[H(x) := H(y)]], \epsilon\rangle}$$

Finally the transition of dynamic allocation is as follows:

$$\langle H, x := \text{new}\rangle \rightarrow \langle H[x := n][\bar{f} := \bar{\rho}], \epsilon\rangle$$

where

- $n \in \mathbb{N} \setminus \mathcal{R}_H(V_P, F_P)$ denotes a location not reachable from program variables through navigation expressions over program fields; such a location always exists by the assumption on heaps that $\mathcal{R}_H(V, F)$ is finite,
- $\bar{f}$ is the sequence of program fields F_P,
- $\bar{\rho}$ is a sequence such that for every i: $\rho_i = H(f_i)[n := H(nil)]$.

The extention to sequential composition, if-then-else, while loops and procedure calls is defined in a standard way, which we include here for a complete presentation. In order to define conditional statements and while loops we assume a relation $\models$ satisfying $H \models x = y$ iff $H(x) = H(y)$, and $H \models x \neq y$ iff $H \not\models x = y$.

$$\frac{\langle H, S\rangle \rightarrow \langle H', \epsilon\rangle}{\langle H, S; S'\rangle \rightarrow \langle H', S'\rangle}$$

$$\frac{H \models B}{\langle H, \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}; S\rangle \rightarrow \langle H, S_1; S\rangle}$$

$$\frac{H \not\models B}{\langle H, \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}; S\rangle \to \langle H, S_2; S\rangle}$$

$$\frac{p :: S \in P}{\langle H, p; S'\rangle \to \langle H, S; S'\rangle}$$

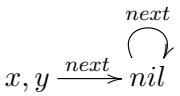
$$\frac{H \models B}{\langle H, \text{while } B \text{ do } S \text{ od}; S'\rangle \to \langle H, S; \text{while } B \text{ do } S \text{ od}; S'\rangle}$$

$$\frac{H \not\models B}{\langle H, \text{while } B \text{ do } S \text{ od}; S'\rangle \to \langle H, S'\rangle}$$

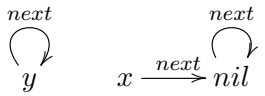
Example 1. The following program inserts an element in front of a linked list with head x:

$$y := \text{new}; y.next := x; x := y$$

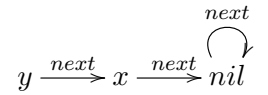
As an illustration of the basic assignments we execute this program on the following (partial representation of a) heap:



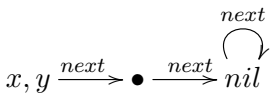
This represents a one-element linked list with head x, and y happens to be aliased with x. Now applying the dynamic allocation statement $y :=$new has the following effect:



Next by execution $x.next := y$ we update the field $next$ of the location to which y points, to point to x:



Finally we execute the statement $x := y$:



This concludes our example.

Chapter 3

A logic for heap abstraction

In this chapter we define our dialect of propositional dynamic logic. The first section introduces its syntax and semantics, together with some examples. We extend in Section 3.2 the programs of the previous chapter with pre- and postconditions to obtain annotated programs.

3.1 Syntax and semantics

In this section we introduce our logic for abstract specification of heap properties, based on propositional dynamic logic. The basic modalities are the fields F, and V is the set of propositional variables. The syntax of dynamic logic formulas is defined in a standard way as follows:

$$\varphi ::= \bot \mid x \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\alpha\rangle\varphi$$
$$\alpha ::= f \mid \alpha;\alpha \mid \alpha + \alpha \mid \alpha^* \mid x? \mid \neg x?$$

where x ranges over V, and f ranges over F. Note that tests occur only on the level of literals. We define $[\alpha]\varphi = \neg\langle\alpha\rangle\neg\varphi$, and use in addition to the above the standard connectives from propositional logic $(\wedge, \rightarrow, \ldots)$. The expressions α are called *navigation expressions*. We define satisfaction of basic formulas φ in a standard way [4]; given a heap H together with a natural number n we have

$$H, n \not\models \bot$$
$$H, n \models x \text{ iff } H(x) = n$$
$$H, n \models \phi_1 \vee \phi_2 \text{ iff } H, n \models \phi_1 \text{ or } H, n \models \phi_2$$
$$H, n \models \neg\phi \text{ iff } H, n \not\models \phi$$
$$H, n \models \langle\alpha\rangle\phi \text{ iff } \exists m \in \mathbb{N} \text{ such that } (n, m) \in H(\alpha) \text{ and } H, m \models \phi$$

where (the relation) $H(\alpha)$ is the extension of $H(f)$ to arbitrary expressions α, defined by structural induction in the standard manner:

$$H(\alpha_1;\alpha_2) = H(\alpha_1) \circ H(\alpha_2)$$
$$H(\alpha_1 + \alpha_2) = H(\alpha_1) + H(\alpha_2)$$
$$H(\alpha^*) = \bigcup_{n\in\mathbb{N}} H(\alpha)^n$$

A variable or field z is *fresh* in a formula φ if it does not occur in φ. We define $\mathit{Var}(\varphi)$ as the set of all variables which occur in φ, and $\mathit{Field}(\varphi)$ as the set of all fields occuring in φ.

As a first attempt for a notion of validity we define in the usual way $H \models \varphi$ iff $H, n \models \varphi$ for all $n \in \mathbb{N}$. However this forces the semantics of formulas also to take into account properties of garbage, i.e., unreachable parts of the heap; this we can solve be redefining validity as $H \models \varphi$ iff $H, n \models \varphi$ for all $n \in \mathcal{R}_H(V, F)$. Unfortunately, this notion of validity gives rise to a highly complicated check of the corresponding entailment relation $\phi \models \psi$: it requires the construction of counterexample models in which ϕ is satisfied in every state, which is formalized by the following implication [8]:

$$\models [(f_1 + \ldots + f_k)^*]\phi \to \psi$$

where $f_1, \ldots, f_k$ are the basic modalities occuring in ϕ and ψ. In order to make this process more tractable we relativize this general notion of validity to a local view from the variables, by the introduction of so-called *rooted formulas*, given by the following grammar:

$$\Phi ::= @x.\varphi \mid \Phi_1 \wedge \Phi_2$$

Then for a formula $\Phi = @x_1.\varphi_1 \wedge \ldots \wedge @x_n.\varphi_n$ and a heap H we define

$$H \models \Phi \text{ iff } \forall i \leq n : H, H(x_i) \models \varphi_i$$

Now given the above Φ and another rooted formula $\Psi = @y_1.\psi_1 \wedge \ldots \wedge @y_m.\psi_m$, checking the entailment $\Phi \models \Psi$ reduces to the construction of a heap H such that $H \models \Phi$ and $H \models @y_i.\neg\psi_i$ for some $i \leq m$.

Note that we thus have introduced a very restricted use of the binding operator in hybrid logic [4]: we only allow top-level occrrences of this operator. Let us discuss the main features of the logic with some examples.

Example 2. Determinism of fields in heaps leads to the fact that the following entailment holds:

$$@x.\langle f\rangle y \wedge \langle f\rangle z \models @y.z \tag{3.1}$$

Variables are nominals, i.e., true in exactly one world:

$$@x.y \models @y.x \tag{3.2}$$

A formula $@x.\langle(f+g)^*\rangle y$ states that y is reachable from x over fields f and g. For example, the following entailment holds:

$$@x.\langle f^*\rangle y \wedge @y.\langle g\rangle z \models @x.\langle(f+g)^*\rangle z \tag{3.3}$$

but the following does not:

$$@x.\langle f^*\rangle y \wedge @y.\langle g\rangle z \models @x.\langle f^*\rangle z \tag{3.4}$$

Finally, as an example of a linked data structure, some variable x being the head of a (non-circular) linked list is simply modeled as

$$@x.\langle next^*\rangle nil \tag{3.5}$$

Compare this to the formula $@x.\langle next^*\rangle\neg nil$; heaps (with a finite number of reachable states) which satisfy that formula must have a loop somewhere, i.e., have x as the head of a linked list with a loop. To specify that x is the head of a linked list which does not contain y, we can write

$$@x.[next^*](\neg y) \wedge \langle next^*\rangle nil \tag{3.6}$$

We conclude this section with the notion of *substitution* used in the definition of strongest postconditions. Substitution of a variable x for z is denoted $\Phi[z/x]$, and substitution of a field f for a navigation expression $\Phi[\alpha/f]$. Both are defined by structural induction in the standard manner; the base cases are as follows:

$$y[z/x] = \begin{cases} z & \text{if } x = y \\ y & \text{otherwise} \end{cases} \qquad g[\alpha/f] = \begin{cases} \alpha & \text{if } f = g \\ g & \text{otherwise} \end{cases}$$

Of pariticular interest is the case of the @ operator:

$$(@y.\varphi)[z/x] = @(y[z/x]).(\varphi[z/x]) \qquad (@y.\varphi)[\alpha/f] = @y.(\varphi[\alpha/f])$$

The relation between substitution in formulas and allocation in heaps is formalized in the following adaptation of the standard substitution lemma:

Lemma 1 (Substitution)**.** *Let H be a heap, Φ a rooted formula, x, z variables, f a field and α a navigation expression such that $H(\alpha)$ is a function $H(\alpha) : \mathbb{N} \to \mathbb{N}$. Then*

1. $H \models \Phi[z/x]$ *iff* $H[x := H(z)] \models \Phi$

2. $H \models \Phi[\alpha/f]$ *iff* $H[f := H(\alpha)] \models \Phi$

Since we allow substitution of general navigation expressions for fields, in order for the update $H[f := H(\alpha)]$ to be well-defined we require that $H(\alpha)$ is deterministic.

3.2 Annotated programs

Given the above logic, we now extend our programs with specifications of pre- and postconditions, by defining an *annotated program* P as follows:

$$P = \{\{\Phi_1\}p_1 :: S_1\{\Psi_1\}, \ldots, \{\Phi_n\}p_n :: S_n\{\Psi_n\}\}$$

where each precondition Φ_i and each postcondition Ψ_i is a rooted formula over variables of V_P and fields of F_P. Statements are defined as before with the exception of while loops, which we assume to be equipped with invariants:

$$(inv : \Phi)\text{while } B \text{ do } S \text{ od}$$

We turn to the definition of correctness of programs with respect to their annotations:

Definition 1 (Partial correctness)**.** *Given a statement S, precondition Φ and postcondition Ψ we define a correctness triple $\models \{\Phi\}S\{\Psi\}$ to be* valid *if for all heaps H:*

$$H \models \Phi \text{ and } \langle H, S\rangle \rightarrow^* \langle H', \epsilon\rangle \text{ implies } H' \models \Psi$$

A program P is correct*, denoted $\models P$, if for every $\{\Phi\}p :: S\{\Psi\} \in P$:*

$$\models \{\Phi\}S\{\Psi\}$$

Note that the standard rules of Hoare logic with respect to the program constructs hold for the above definition of partial correctness.

Example 3. We give as example the following annotation of an implementation of an insertion into a (non-circular) linked list.

$$\{@x.\langle next^*\rangle nil\}y := \text{new}; y.next := x; x := y\{@x.\langle next^*\rangle nil\} \tag{3.7}$$

Remember from Example 2 that a (non-circular) linked list with head x is represented succinctly by the formula $@x.\langle next^*\rangle nil$. Note that according to

this annotation the property of non-circularity is preserved by the insertion of a new element. On the other hand, the annotated program

$$\{@x.\langle next^* \rangle nil\} y.next := x; x := y\{@x.\langle next^* \rangle nil\} \tag{3.8}$$

is clearly incorrect because if y itself is already part of the list then its insertion will introduce a circularity, as z sets the $next$ field to point to x.

Chapter 4

Deciding entailment

Automated checking of the entailment relation of our logic turns out to be a crucial to our approach. In this chapter we discuss a method for deciding entailment which is based on classical tableaux methods for dynamic logic, but deals with the specific characteristics of our restricted class of models. This method is detailed in the first section, and in the second section the output of running examples on the actual Maude implementation is given.

4.1 A modular approach

In this section we sketch the main characteristics of our tableaux method for deciding entailment between rooted formulas and its prototype implementation in the rewrite logic engine of Maude. To check an entailment $\Phi \models \Psi$ we search for a counterexample, that is, a heap which satisfies Φ but does not satisfy Ψ. In our case, for a given $\Phi = @x_1.\varphi_1 \wedge \ldots \wedge @x_n.\varphi_n$ and $\Psi = @y_1.\psi_1 \wedge \ldots \wedge @y_m.\psi_m$ this means the construction of a heap H such $H \models \Phi$ and $H \models @y_i.\neg\psi_i$ for some $i \leq m$.

To this end we introduce in Maude the sorts *World*, *Model*, *NewWorld* and *Config*. A term of sort *World* is constructed by the operation

$$op\ \_.\_\ :\ Nat\ Form \rightarrow World$$

where $Form$ denotes the sort of dynamic logic formulas. The sort of natural numbers Nat is used to identify "worlds", as we will call such terms in the sequel. A heap is represented by a term of sort *Model* which is constructed by

$$op\ \_;\_;\_\ :\ Worlds\ Transitions\ Nat \rightarrow Model$$

where *Worlds* denotes a sequence of terms of sort *World*, a term of sort *Transitions* denotes a set of labeled transitions between worlds represented by their natural numbers. For technical convenience *Nat* is used to represent a bound on the number of worlds in the model. The standard (structural) tableaux rules for propositional dynamic logic operate on the terms of the sort *NewWorlds* constructed by

$$op\ [\_,\_,\_,\_,\_]\ :\ Nat\ Field\ Form\ Form\ Next \rightarrow NewWorld$$

where

1. the first argument represents its originating world,
2. the second argument represents the transition from that world to this new world "under construction",
3. the third argument is a conjunction of formulas which are supposed to hold in this new world but are still to be processed, whereas
4. the fourth argument represents a conjunction of formulas which are already processed and therefore do hold., finally
5. the last argument *Next* is a set of formulas labeled by fields, indicating the worlds supposed to originate from this one.

Finally a term of sort *Config* is constructed by

$$op\ (\_|\_)\ :\ NewWorlds\ Model \rightarrow Config$$

$$op\ \_|\_\ :\ Config\ Config \rightarrow Config$$

where *NewWorlds* is denotes a sequence of worlds of terms of sort *NewWorld*. A term of sort *Config* thus consists of a list of pairs, each of which represents a set of new worlds which are to be finalized and a term of sort *Model* which contains *finalized* worlds.

For a proper treatment of the identification of nominals our tableaux method operates on a list of configurations. For example, a disjunction $\varphi \vee \psi$ gives rise to a *split* of the current configuration into two new configurations corresponding to the two disjuncts φ and ψ, respectively. The entailment relation $\Phi \models \Psi$ (Φ and Ψ as above) is therefore represented by a sequence of configurations $C_1, \ldots C_m$ where each C_i consists of an empty model and a set of worlds under construction representing the formulas:

$$x_1 \wedge \varphi_1, \ldots, x_n \wedge \varphi_n, y_i \wedge \neg\psi_i$$

Our method distinguishes the following three (disjoint) sets of rewrite rules to be applied to indivudual configurations.

1. First the standard (structural) tableaux rules for dynamic logic are applied on the worlds under construction. A literal conjunct for example is simply transferred to the set of processed formulas. As described above, a disjunction splits the current configuration into two. A formula $\langle\alpha^*\rangle\varphi$ at this stage is treated as a disjunction $\varphi \vee \langle\alpha\rangle\langle\alpha^*\rangle\varphi$. A formula $\langle f\rangle\varphi$ is transferred both to the processed formulas and to the set of next worlds. A formula $[f]\varphi$ is treated similarly (note that f is deterministic). All other formulas are dealt with by the usual reduction axioms of dynamic axioms.

2. The second phase consists of the *merging* of finalized worlds containing the same nominal, and its propagation to the transitions to maintain the deterministic nature of fields.

3. The final phase consists of a *star track*, which checks if the eventualities of the form $\langle\alpha^*\rangle\varphi$ are indeed validated. This phase is implemented by computing α^* using the given bound on the number of worlds.

Every configuration which passes through these three phases and the consistency check on finalized worlds is a valid counterexample. Other configurations are deleted. We denote by $\Rightarrow_T$ the consecutive application of the above sets of rewrite rules. We have the following main theorem.

Theorem 1. *Let Φ and Ψ be rooted formulas as defined above. Further let $C_1, \ldots, C_m$ be a corresponding sequence of configurations, also as defined above. We have*

$$\Phi \models \Psi \text{ iff } C_1, \ldots, C_m \Rightarrow_T^* \epsilon$$

where $\Rightarrow_T^*$ denotes the transitive closure of $\Rightarrow_T$ and ϵ denotes the empty list.

The consecutive application of the above three (disjoint) sets of rewrite rules, i.e., (1) standard structural rules, (2) propagation of nominal identification and (3) checking eventualites, yields a modular method which allows for a transparant proof. We only observe here that at the heart of the above theorem lies the basic fact that the consecutive application of the above sets of rewrite rules is indeed correct because of the deterministic interpretation of the field names. For example, adding the converse operator on fields, i.e., f^{-1}, merging two worlds (because of a common nominal) would in general require a renewed validation (and corresponding propagation) of a formula $[f^{-1}]\phi$ of the resulting world. On the other hand, a termination proof for our tableaux method *requires* this consecutive application of the above rewrite rules. Intuitively, no rule in the above first set of (structural) rewrite rules is

applicable (to a term of sort *Config*) when all terms of sort *NewWorld* have been processed, i.e., transformed into terms of sort *World*. Distinguishing, in a configuration, between new worlds and finalized worlds allows a simple check whether a processed new world already appears as a final world in the model. In case the processed world already exists we only need to possibly add a new transition (from the originating world to the corresponding final world). Nominal identifaction at the first phase on finalized worlds may lead to divergence. We illustrate this with an informal example.

Example 4. Suppose in a given configuration there exists a final world $\varphi \wedge x$ for some φ. Furthermore suppose there exists a world under construction in which the formula $[f^*]x$ is to be made true. Processing this formula leads to the introduction of a final world in which $x \wedge [f][f^*]x$ is true (assuming that such a world does not yet exist) and a new world orginating from this fresh final world in which again $[f^*]x$ is to be made true. Because of the nominal x occuring both in $\varphi \wedge x$ and $x \wedge [f][f^*]x$ these worlds then can be identified. As a consequence the above scenario will repeat itself.

4.2 Running the entailment checker

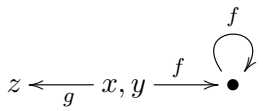
We recall two examples from Chapter 3 and apply the entailment checker described in the previous section. Example 3.3 has the following output when run in the Maude interpreter:

```
rewrite in TABLO :
  ((x . < f * > y),y . < g > z) |= x . < (f + g) * > z .
rewrites: 1444 in 8ms cpu (10ms real) (168082 rewrites/second)
result DelimConfig: {empty}
```

which tells us no counterexample could be found, and thus that the entailment holds. On the contrary for Example 3.4 a counterexample is generated:

```
rewrite in TABLO : ((x . < f * > y),y . < g > z) |= x . < f * > z .
rewrites: 312 in 0ms cpu (1ms real) (355353 rewrites/second)
result DelimConfig:
 {(true ; (0 . x & y & - z & (< g > z) & (< f * > y) &
 ([f][f *]- z) & [f *]- z),(1 . - z & ([f][f *]- z) & [f *]- z),
 3 . z; < f,0,1 >,< f,1,1 >,< g,0,3 > ; 4),(...)}
```

The above output represents a model which we can graphically represent as follows:



Indeed this model satisfies $@x.\langle f^* \rangle y$ and $@y.\langle g \rangle z$ but not $@x.\langle f^* \rangle z$.

Chapter 5

Verification

The first section of this chapter introduces the strongest postconditions of assignments. Then in the second section, these strongest postconditions are extended to a verification condition generator for full programs, which is combined with the entailment checker, resulting in our method for automated verification. In the third section we shortly discuss the overall structure of the Maude implementation, and finally in the last section we provide and discuss the output of checking an example program in our automated verification tool.

5.1 Strongest postconditions of assignments

In general, verification of annotated programs is based on the generation of verification conditions. This can be done systematically by computing weakest preconditions or strongest postconditions. It is worthwhile to observe that it is unclear and highly problematic how to generate statically the weakest precondition in our logic. For instance, the weakest precondition of $x := y.f$ with respect to the rooted formula $@z.x$ is $@y.\langle f \rangle z$. Therefore, our approach is based on computing strongest postconditions, which in contrast does allow a relatively simple characterization based on substitution. However, the standard way of expressing the strongest postconditions requires the introduction of existential quantification to denote the old value of the updated variable. In our case this would give rise to an undecidable logic. However in general (top-level) existential quantifiers can be eliminated in the entailment relation. To show this we first extend our logic with top-level existential quantification:

$$\Phi_E = \exists z.\Phi_E \mid \exists g.\Phi_E \mid \Phi$$

where Φ is a rooted formula, z is a variable and g is a field.

With $FVar(\Phi_E)$ we denote all variables not bound by a quantifier, and similarly with $FField(\Phi_E)$ all such fields. The semantics is defined as follows:

$$H \models \exists z.\Phi \text{ iff } H[z := m] \models \Phi \text{ for some } m \in \mathbb{N}$$
$$H \models \exists g.\Phi \text{ iff } H[g := \rho] \models \Phi \text{ for some } \rho : \mathbb{N} \to \mathbb{N}$$

Now we have the following basic logical equivalence, relating existentially quantified variables and fields in the antecedent to (implicitly) universally quantified variables in an entailment relation:

Lemma 2. *Let Φ and Ψ be formulas. For any $z \in V \setminus FVar(\Psi)$:*

$$\exists z.\Phi \models \Psi \textit{ iff } \Phi \models \Psi$$

For any $g \in F \setminus FField(\Psi)$

$$\exists g.\Phi \models \Psi \textit{ iff } \Phi \models \Psi$$

Proof. We treat the case of a variable. Suppose $\exists z.\Phi \models \Psi$ and $H \models \Phi$ for some heap H. Then since $H = H[z := H(z)]$ by definition $H \models \exists z.\Phi$ and so $H \models \Psi$ by assumption.

Conversely suppose $\Phi \models \Psi$ and $H \models \exists z.\Phi$. Then $H[z := n] \models \Phi$ for some $n \in \mathbb{N}$, so by assumption $H[z := n] \models \Psi$. But since z does not occur free in Ψ, it is easy to show that $H \models \Psi$. □

For the purpose of generating verification conditions, described in the following section, the above lemma justifies the introduction of fresh variables in the strongest postcondition *without* existential quantification. The strongest postcondition $SP(x := y, \Phi)$ of a variable assignment of the form $x := y$ and a formula Φ therefore is given by the formula

$$\Phi[z/x] \wedge @y[z/x].x$$

where z is a fresh variable, not occuring in Φ or V_P. In the special case of an assignment $x := x$, we should have that z equals x, which is indeed taken care of by the substitution in $@x[z/x].y$. The strongest postcondition $SP(x := y.f, \Phi)$ of a variable assignment $x := y.f$ and a formula Φ is similar to the above update $x := y$:

$$\Phi[z/x] \wedge @y[z/x].\langle f \rangle x$$

where z is as again a fresh variable. The strongest postcondition $SP(x.f := y, \Phi)$ of an assignment $x.f := y$ and a formula Φ is given by the formula

$$\Phi[((x?; g) + (\neg x?; f))/f] \wedge @x.(\neg nil \wedge \langle f \rangle y)$$

where g is a fresh field name not occuring in Φ or F_P. Here g represents the old value of f. The formula $@x.\neg nil$ is required to match the operational semantics, which states that field update to an alias of *nil* is not allowed. For example when $\Phi = @x.nil$, the above strongest postcondition is a contradiction, which is correct, since any execution of $x.f := y$ on a heap satisfying Φ would block according to the operational semantics. Finally the strongest postcondition $SP(x := \text{new}, \Phi)$ of dynamic allocation $x := \text{new}$ and a formula Φ is given by the formula

$$\Phi[z/x] \wedge \bigwedge_{f \in F_P} (@x.\langle f \rangle nil) \wedge \bigwedge_{v \in V_P[z/x]} (@v.[(f_1 + \ldots + f_k)^*]\neg x)$$

where $V_P[z/x] = V_P \setminus \{x\} \cup \{z\}$ and $\{f_1, \ldots, f_k\} = F_P$. Intuitively the above formula states that x is unreachable from any other program variable after being allocated, and that its fields are intialized to point to *nil*.

Example 5. We illustrate the above in terms of insertion into a non-circular list of Example 3. For notational convenience we assume that $F_P = \{next\}$ and $V_P = \{y, x\}$. We first observe that $SP(y := \text{new}, @x.\langle next^* \rangle nil)$ equals

$$@x.\langle next^* \rangle nil \wedge @y.\langle next \rangle nil \wedge @x.[next^*]\neg y \wedge @z.[next^*]\neg y$$

Next we compute the strongest postcondition of the above formula for the assignment $y.next := x$:

$$@x.\langle \pi^* \rangle nil \wedge @y.\langle \pi \rangle nil \wedge @x.[\pi^*]\neg y \wedge @z.[\pi^*]\neg y \wedge @y.(\neg nil \wedge \langle next \rangle x)$$

where π stands for $(y?; f + \neg y?; next)$, for some new field name f. Finally, we compute the strongest postcondition of this latter formula for the assignment $x := y$:

$$@w.\langle \pi^* \rangle nil \wedge @y.\langle \pi \rangle nil \wedge @w.[\pi^*]\neg y \wedge @z.[\pi^*]\neg y \wedge @y.(\neg nil \wedge \langle next \rangle w) \wedge @y.x$$

In Section 5.4 we show that the prototype implementation of our semantic tableaux method reports that the resulting strongest postcondition entails $@x.\langle next^* \rangle nil$. For now we give an informal argument as to why this is the case. To this end, suppose the above strongest postcondition is true in some heap H. First note that π agrees with *next* on every world where y does not

hold. Thus since $@w.[\pi^*]\neg y$ holds it follows by induction that $@w.[next^*]\neg y$ does, too; but then from $@w.\langle\pi^*\rangle nil$ we may conclude $@w.\langle next^*\rangle nil$, which again can be shown inductively. So w is the head of a linked list; and since $@y.\langle next\rangle w$ holds y is, too. Finally $@y.x$ states that x and y are equal, and so $@x.\langle next^*\rangle nil$ as desired.

We conclude this section with the following theorem, stating that the above strongest postconditions are sound and complete with respect to the operational semantics:

Theorem 2. *For every formula Φ:*

$$SP(A,\Phi) \models \Psi \quad \textit{iff} \quad \models \{\Phi\}A\{\Psi\}$$

where A is an assignment, $Var(\Psi) \subseteq V_P \cup Var(\Phi)$ and $Field(\Psi) \subseteq F_P \cup Field(\Phi)$.

The proof of this theorem is omitted here to preserve the flow of reading, but can be found in Appendix A.

5.2 Automated verification

We turn to our method for automated verification, which is based on generating verification conditions using the above syntactic descriptions of the strongest postconditions of assignments, and then proving or disproving the generated conditions using the tableaux method of Chapter 4. Our method is implemented in terms of a rewrite system in Maude; we give here a high-level description. This system operates on a sequence of annotated statements or procedure declarations, i.e., a sequence of triples of the form $\{\Phi\}S\{\Psi\}$ or $\{\Phi\}p :: S\{\Psi\}$. Now rewriting a triple $\{\Phi\}S\{\Psi\}$ can be thought of as abstract execution of S, starting in the symbolic state Φ. We show how to treat each of the programming language constructs. Procedure declarations of the form $\{\Phi\}p :: S\{\Psi\}$ are rewritten as follows:

$$\{\Phi\}p :: S\{\Psi\} \Rightarrow \{\Phi\}S;\epsilon\{\Psi\} \tag{5.1}$$

Here ϵ allows for an easy treatment of sequential composition. For any assignment A, we simply compute the strongest postcondition as given above:

$$\{\Phi\}A; S\{\Psi\} \Rightarrow \{SP(A,\Phi)\}S\{\Psi\} \tag{5.2}$$

On an if-then-else statement with condition B, we branch into two threads of execution, one where B is true and one where it is not. Note that B is

either an equality or a disequality and thus not strictly a formula in our logic; with $\widehat{B}$ we denote a logical formula corresponding to the expression B, which is defined simply as $\widehat{B} = @x.y$ in case B is of the form $x = y$, and $@x.\neg y$, in case it is of the form $x \neq y$. Thus we obtain the following.

$$\{\Phi\}\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}; S\{\Psi\} \Rightarrow \{\Phi \wedge \widehat{B}\}S_1; S\{\Psi\}, \{\Phi \wedge \widehat{\neg B}\}S_2; S\{\Psi\} \tag{5.3}$$

A while loop is treated by proving that the invariant is preserved on execution of the body and continuing after the loop with the invariant and the negated loop condition:

$$\frac{\Phi \models \Theta}{\{\Phi\}(inv : \Theta)\text{while } B \text{ do } S \text{ od}; S'\{\Psi\} \Rightarrow \{\Theta \wedge \widehat{B}\}S\{\Theta\}, \{\Theta \wedge \widehat{\neg B}\}S'\{\Psi\}} \tag{5.4}$$

On procedure call, we show that the current state entails the precondition of the callee, and then continue with its postcondition.

$$\frac{\{\Phi_i\}p_i :: S_i\{\Psi_i\} \in P \quad \Phi \models \Phi_i}{\{\Phi\}p_i; S\{\Psi\} \Rightarrow \{\Psi_i\}S\{\Psi\}} \tag{5.5}$$

Finally we terminate succesfully when the current state entails the postcondition:

$$\frac{\Phi \models \Psi}{\{\Phi\}\epsilon\{\Psi\} \Rightarrow \top} \tag{5.6}$$

The above (sets of) rules are combined into a relation $\Rightarrow$, which is lifted to sets of triples of the form $\{t_1, \ldots, t_n\}$, where each t_i is either of the form $\{\Phi\}S_i\{\Psi\}$ or $\{\Phi\}p_i :: S_i\{\Psi\}$.

Definition 2. *Let $\Rightarrow^*$ denote the transitive closure of $\Rightarrow$. We define*

$$\vdash \{\Phi\}S\{\Psi\} \textit{ iff } \{\{\Phi\}S\{\Psi\}\} \Rightarrow^* \{\top\}$$

For an annotated program $P = \{\{\Phi_1\}p_1 :: S_1\{\Psi_1\}, \ldots, \{\Phi_n\}p_n :: S_n\{\Psi_n\}\}$ we define

$$\vdash P \textit{ iff } \{P\} \Rightarrow^* \{\top\}$$

We proceed to discuss the correctness of the above approach. Unfortunately, we can not generalize Theorem 2 to statements S as it is impossible to achieve completeness. The reason for this is that given the combination of a decidable logic for assertions and a Turing-complete programming language, there is no complete Hoare-style axiom system [6]. We continue to show that our method is sound. A technical difficulty is that for basic assignments $\models \{\Phi\}A\{SP(A, \Phi)\}$ does not hold, since $SP(A, \Phi)$ introduces a

fresh variable or field. To overcome this problem we first define an alternative version SP_E of the strongest postconditions of assignments as follows. For any A of type $x := y$, $x := y.f$ or $x :=$ new:

$$SP_E(A, \Phi) = \exists z.SP(A, \Phi)$$

where z is the fresh variable introduced in $SP(S, \Phi)$. Similarly we define

$$SP_E(x.f := y, \Phi) = \exists g.SP(S, \Phi)$$

Now by Lemma 2 and Theorem 2 we have the following:

Corollary 1. *For any formula Φ and assignment A:*

$$\models \{\Phi\}A\{SP_E(A, \Phi)\}$$

We are now ready to prove the correctness of the method:

Theorem 3 (Soundness)**.** *For any program P: if $\vdash P$, then $\models P$.*

Proof. We prove that for any triple $\{\Phi\}S\{\Psi\}$ such that $Var(\Psi) \subseteq V_P$ and $Field(\Psi) \subseteq F_P$:

$$\{\{\Phi\}S; \epsilon\{\Psi\}\} \Rightarrow^* \{\top\} \text{ implies } \models \{\Phi\}S\{\Psi\}$$

by induction on the size of statements. The base case, for the empty statement, follows directly from the definition of correctness triples and rule (5.6) defining $\Rightarrow$, given that the tableaux method is correct (Theorem 1). Next let $S; S'$ be a statement such that S is not a composition, $\{\Phi\}S; S'; \epsilon\{\Psi\} \Rightarrow^* \top$ and suppose our claim holds for any statement smaller than $S; S'$. We treat the case that S is an assignment and the case that S is an if-then-else statement.

- Suppose S is an assignment. Then $\{\Phi\}S; S'; \epsilon\{\Psi\}$ can only be rewritten by rule (5.2), so $\{\{SP(S, \Phi)\}S'\{\Psi\}\} \Rightarrow^* \{\top\}$ and since S' is smaller than S, by the induction hypothesis $\models \{SP(S, \Phi)\}S'\{\Psi\}$. Since $Var(\Psi) \subseteq V_P$ and $Field(\Psi) \subseteq F_P$ but the introduced existentially quantified variable or field in $SP_E(S, \Phi)$ is not in V_P or F_P respectively, it is easy to show that $\models \{SP_E(S, \Phi)\}S'\{\Psi\}$ holds – this is in fact a generalized version of a standard rule in Hoare logic for the introduction of existential quantifiers. Further by Corollary 1 we have $\models \{\Phi\}S\{SP_E(S, \Phi)\}$. It follows that $\models \{\Phi\}S; S'\{\Psi\}$ holds.

- Suppose $S =$ if B then S_1 else S_2 fi for some S_1, S_2 and B. Then rule (5.3) is applied, so both $\{\{\Phi \wedge \widehat{B}\}S_1; S\{\Psi\}\} \Rightarrow^* \{\top\}$ and $\{\{\Phi \wedge \widehat{\neg B}\}S_2; S\{\Psi\}\} \Rightarrow^* \{\top\}$. Now by the induction hypothesis we have $\models \{\Phi \wedge \widehat{B}\}S_1; S\{\Psi\}$ and $\models \{\Phi \wedge \widehat{\neg B}\}S_2; S\{\Psi\}$, and it easily follows that $\models \{\Phi\}$if B then S_1 else S_2 fi$; S\{\Psi\}$ holds.

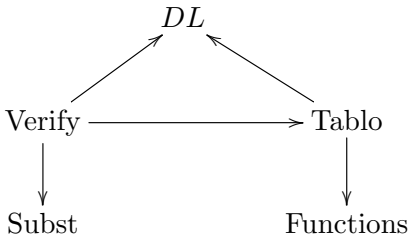
□

5.3 Structure of the implementation

In this section we shortly discuss the overall structure of our Maude implementation. We recall that the main components are the automated verification generator and the tableaux method. These different components are structured into the following interdependent *modules*:

- *Verify*: The Verify module contains the definition of the programming language, and the implementation of the automated verification engine.
- *Tablo*: This module contains the rewrite theory implementing our tableaux method.
- *DL*: An algebra of formulas propositional dynamic logic is described in this module, together with a set of reduction axioms which allow to put any formula in a normal form.
- *Functions*: This is a helper module for the tableaux method, in particular supporting the reachability check for eventualitites and substitution of transitions.
- *Subst*: The *Subst* module supplies substitution of variables and fields, as introduced in Section 3.1.

The dependencies between the different modules are visualized in the following diagram:



An arrow from a module A to a module B denotes a dependency. In this setting this means, for example, that *Tablo* can only work in the presence of the theories *DL* and *Subst*. Thus the possible order of loading these modules is obtained by taking any reverse topological sorting of the above diagram.

5.4 Running the verifier

In this section we provide the output of running some of the introduced examples on our Maude implementation of the tableaux method and the automated verification.

As an example of the full implementation of the automated verification based on generating verification conditions and proving them using the entailment checker we try to verify first the incorrect list insertion (Example 3.8):

```
rewrite in VERIFY :
 ver(m1 . x.< next* >nil . ((y..next):=x); x:=y . x.< next* >nil).
rewrites: 53413 in 91ms cpu (92ms real) (582050 rewrites/second)
result PConfig: (...) {(true ; (0 . x & y & - nil & x ' & (...)
 3 . nil & < ((y ; next') + (- y ; next)) * > nil ;
 < next,0,0 >,< next',0,3 > ; 7),(...)
```

This returns as a counterexample a model in which x points to a circular list; in fact, $x.next = x$. Interesting to see, however, is that $next'$, the fresh field introduced to represent *next* before the assignment $y.next := x$, does map world 0 to world 3 (in which *nil* holds); thus, we can see that $y.next := x$ was exactly the statement introducing the circularity.

The corrected version (Example 3.7) is indeed verified, as expected:

```
rewrite in VERIFY :
  ver(m1 . x.< next * > nil . (y := new[next | x y]) ; ((y .. next) := x)
  ; x := y . x . < next * > nil) .
rewrites: 13378832 in 25262ms cpu (25270ms real)
 (529585 rewrites/second)
result PConfig: success
```

Chapter 6

Future directions

In this concluding chapter we propose several directions for future research. One interesting thread of research is the optimization of the tableaux method for entailment. There are several options to explore. First, a straightforward but non-trivial exercise would be the implementation into a fast imperative language. Secondly, the deterministic nature of fields should allow for huge improvements. For example in most (but unfortunately not all) cases it is sound to generalize the rule for detection of new worlds which already exist – currently a new world φ is added if φ does not yet occur in the model – to a rule which only adds such a world if there exist no world which *subsumes* it; i.e., if there already exists a world $\varphi \wedge \psi$ in the model, the new world is not added. Unfortunately this approach is in general not sound. It would be interesting to identify precisely the problematic cases, so that we can apply the optimized rule whenever it is safe to do so.

The introduction of implicitly exisistentially quantified variables in the strongest postconditions increases the size of the formulas. Possibly these logical variables can be eliminated using an adapted version of the tableaux method.

Yet another question is whether there exists a sound and complete axiomatization of our heap logic. In particular, is it possible to axiomatize the behaviour of nominals? This would open possibilities for algebraic reasoning about equivalence of formulas.

Finally, our language does not support local variables. In our setting full recursion with local variables requires adaptation rules suitable for automated reasoning [11].

Bibliography

[1] Martin Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In Nachum Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 11–41. Springer, 2003.

[2] K. R. Apt, F. S. de Boer, and E.-R. Olderog. *Verification of Sequential and Concurrent Programs, 3rd Edition*. Texts in Computer Science. Springer-Verlag, 2009. 502 pp, ISBN 978-1-84882-744-8.

[3] M. Benedikt, T.W. Reps, and S. Sagiv. A decidable logic for describing linked data structures. In *Proceedings of the 8th European Symposium on Programming Languages and Systems*, ESOP '99, pages 2–19, London, UK, 1999. Springer-Verlag.

[4] P. Blackburn, M. de Rijke, and Y. Venema. *Modal logic*. Cambridge University Press, 2001.

[5] J. Brotherston and M. I. Kanovich. Undecidability of propositional separation logic and its neighbours. In *LICS*, pages 130–139, 2010.

[6] E.M. Clarke. Programming language constructs for which it is impossible to obtain good hoare-like axioms. volume 26 of *Journal of the ACM*, pages 126–147, 1979.

[7] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of maude. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.

[8] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press. 2000.

[9] U. Hustadt and R.A. Schmidt. A comparison of solvers for propositional dynamic logic. In B. Konev, Renate A. Schmidt, and Stephan Schulz,

editors, *Proceedings of the Workshop on Practical Aspect of Automated Reasoning (PAAR-2010)*, 2010.

[10] M. Kaminski and G. Smolka. Terminating tableaux for hybrid logic with eventualities. In Jürgen Giesl and Reiner Hähnle, editors, *IJCAR 2010*, volume 6173 of *LNCS (LNAI)*, pages 240–254. Springer, Jul 2010.

[11] D. A. Naumann. Calculating sharp adaptation rules. *Information Processing Letters*, 77:2001, 2000.

[12] J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74. IEEE Computer Society, 2002.

[13] U. Sattler and M. Y. Vardi. The hybrid μ-calculus. In *Proceedings of the First International Joint Conference on Automated Reasoning*, IJCAR '01, pages 76–91, London, UK, UK, 2001. Springer-Verlag.

[14] Y. Yuasa Y. Tanabe, T. Sekizawa and K. Takahashi. Pre- and post-conditions expressed in variants of the modal μ-calculus. *IEICE Transactions*, pages 995–1002, 2009.

Appendix A

Full proofs

Theorem 2. Let Φ be a formula, A a heap update, and Ψ a formula such that $\mathit{Var}(\Psi) \subseteq V_P \cup \mathit{Var}(\Phi)$ and $\mathit{Field}(\Psi) \subseteq F_P \cup \mathit{Field}(\Phi)$. We do not treat assignments of the form $x := y$, as their proof is analogous to the case $x := y.f$.

Assume $SP(A, \Phi) \models \Psi$, and let H be a heap such that $H \models \Phi$. To prove is that if $\langle H, A\rangle \to \langle H', \epsilon\rangle$, then $H' \models \Psi$. We continue by cases on A.

1. (A = *x:=y.f*) *To show*: $H[x := H(f)(H(y))] \models \Psi$.

   Let z be the fresh variable introduced in $SP(x := y.f, \Phi)$, then

   $$H[z := H(x)] \models \Phi[z/x]$$

   since z does not occur in Φ. Let

   $$H' = H[z := H(x)][x := H(f)(H(y))]$$

   Then $H' \models \Phi[z/x]$ since x does not occur in $\Phi[z/x]$.

   Next we distinguish two cases. If $x = y$ we have $H(y) = H(x) = H[z := H(x)](z) = H'(z)$ so

   $$H'(f)(H'(z)) = H'(f)(H(y)) = H(f)(H(y)) = H'(x)$$

   Otherwise if $x \neq y$, then $H'(y) = H(y)$ so

   $$H'(f)(H'(y)) = H'(f)(H(y)) = H'(x)$$

   It follows that $H' \models @y[z/x].\langle f\rangle x$ holds in both cases. Thus $H' \models SP(S, \Phi)$ and by assumption $H' \models \Psi$. Since z does not occur in V_P or $\mathit{Var}(\Phi)$, by assumption it does not occur in Ψ, and it follows that $H[x := H(f)(H(y))] \models \Psi$.

2. $(A = x.f := y)$ If $H(x) = H(nil)$ then $\langle H, x.f := y\rangle$ blocks. Thus we continue to prove that $H[f := H(f)[H(x) := H(y)]] \models \Psi$ holds whenever $H(x) \neq H(nil)$.

   Assume $H(x) \neq H(nil)$. Let g be the fresh field of $SP(x.f := y, \Phi)$ not occuring in Φ. Then

   $$H[g := H(f)] \models \Phi[((x?; g) + (\neg x?; f))/f]$$

   Let $H' = H[g := H(f)][f := H(f)[H(x) := H(y)]]$. Since changing the value of $H(f)$ at $H(x)$ does not affect the truth of $\Phi[((x?; g) + (\neg x?; f))/f]$ in $H[g := H(f)]$, we have

   $$H' \models \Phi[((x?; g) + (\neg x?; f))/f]$$

   Further $H' \models @x.\langle f\rangle y$ and by assumption $H \models @x.\neg nil$ from which it follows that $H' \models @x.\neg nil$. Thus $H' \models SP(S, \Phi)$, and again by assumption $H' \models \Psi$. Since g does not occur in F_P or $Field(\Phi)$, it does not occur in Ψ, and we have $H[f := H(f)[H(x) := H(y)]] \models \Psi$.

3. $(A = x := \text{new})$ *To show:* $H[x := n][\bar{f} := \bar{\rho}] \models \Psi$ with n and $\bar{\rho}$ defined as above.

   Let z be the fresh variable introduced in $SP(x := \text{new}, \Phi)$, then $H[z := H(x)] \models \Phi[z/x]$. Now let $n \in \mathbb{N} \setminus \mathcal{R}_H(V_P, F_P)$, let $\bar{f}$ be the sequence of program fields F_P and let $\bar{\rho}$ be a sequence of the same size such that for all i: $\rho_i = H(f_i)[n := H(nil)]$. Let

   $$H' = H[z := H(x)][x := n][\bar{f} := \bar{\rho}]$$

   We have $H' \models \Phi[z/x]$ since x does not occur in $\Phi[z/x]$. By definition n is unreachable from every $v \in V_P$ over fields from F_P, which is simply formalized as

   $$H \models \bigwedge_{v \in V} v \to [(f_1 + \ldots + f_k)^*]\neg x$$

   where $\{f_1, \ldots, f_k\} = F_P$. Now since in H', z is assigned the value of x and the fields F_P are only changed at n, we have

   $$H' \models \bigwedge_{v \in V[z/x]} v \to [(f_1 + \ldots + f_k)^*]\neg x$$

   Finally $H' \models \bigwedge_{f \in F_P}(@x.\langle f\rangle nil)$ by definition of $\bar{\rho}$. But now $H' \models SP(x := \text{new}, \Phi)$ and so $H' \models \Psi$, and since z does not occur in V_P or $Var(\Phi)$, it does not occur in Ψ, so $H[x := n][\bar{f} := \bar{\rho}] \models \Psi$.

This concludes the soundness proof.

Now assume $\models \{\Phi\}A\{\Psi\}$. We must show that $SP(A, \Phi) \models \Psi$ holds. To this end we take a heap H which models the strongest postcondition, and apply some substitution to it to get a heap H' which models Φ. Then by assumption applying the operation to H' either blocks or yields a heap H'' which models Ψ; we then get our result by showing that H'' is equal to to H, and thus that $H \models \Psi$.

1. $(A = x := y.f)$ Assume $H \models \Phi[z/x] \wedge (@y[z/x].\langle f \rangle x)$. Then by the substitution lemma $H[x := H(z)] \models \Phi$, so by assumption

$$H[x := H(z)][x := H(f)(H(y))] \models \Psi$$

We distinguish two cases. If $x = y$, then $H \models @z.\langle f \rangle x$, so $H(f)(H(z)) = H(x)$, so

$$\begin{aligned} & H[x := H(z)][x := H(f)(H(y))] \\ & = H[x := H(z)][x := H(f)(H(x))] \\ & = H[x := H(f)(H(z))] \\ & = H \end{aligned}$$

Otherwise if $x \neq y$, then $H \models @y.\langle f \rangle x$, so $H(f)(H(y)) = H(x)$, so

$$\begin{aligned} & H[x := H(z)][x := H(f)(H(y))] \\ & = H[x := H(f)(H(y))] \\ & = H \end{aligned}$$

Either way we get $H = H[z := H(x)][x := H(f)(H(y))]$ and so $H \models \Psi$.

2. $(A = x.f := y)$ Assume

$$H \models \Phi[((x?; g) + (\neg x?; f))/f] \wedge (@x.\neg nil \wedge \langle f \rangle y)$$

First note that $((x?; g) + (\neg x?; f))$ is a deterministic field expression, i.e., is a function when evaluated on H:

$$\begin{aligned} & H((x?; g) + (\neg x?; f)) \\ & = \{(n, m) | H(x) = o \wedge H(g)(n) = m\} \cup \{(n, m) | H(x) \neq n \wedge H(f)(n) = m\} \\ & = H(f)[H(x) := H(g)(H(x))] \end{aligned}$$

By the substitution lemma $H[f := H((x?;g)+(\neg x?;f))] \models \Phi$. By the above equality we can rewrite the expression into a field update, let

$$H' = H[f := H(f)[H(x) := H(g)(H(x))]]$$

Then $H' \models \Phi$. As $H \models @x.\neg nil$, also $H' \models @x.\neg nil$ and so $H'(x) \neq H'(nil)$, which implies by definition

$$\langle H', x.f := y\rangle \to \langle H'[f := H'(f)[H'(x) := H'(y)]], \epsilon\rangle$$

and consequently, by assumption,

$$H'[f := H'(f)[H'(x) := H'(y)]] \models \Psi$$

Now let us examine the above substitution:

$$\begin{aligned} H'(f)[H'(x) := H'(y)] &= H(f)[H(x) := H(g)(H(x))][H'(x) := H'(y)] \\ &= H(f)[H(x) := H(g)(H(x))][H(x) := H(y)] \\ &= H(f)[H(x) := H(y)] \end{aligned}$$

so

$$H'[f := H(f)[H(x) := H(y)]] \models \Psi$$

and since the first substitution (of H') is cancelled by the above substitution we have

$$H[f := H(f)[H(x) := H(y)]] \models \Psi$$

Further $H \models @x.\langle f\rangle y$, so we have $H(f)(H(x)) = H(y)$, so

$$H[f := H(f)[H(x) := H(y)]] = H$$

from which we conclude $H \models \Psi$.

3. $(A = x := \text{new})$ Assume

$$H \models \Phi[z/x] \wedge \bigwedge_{f \in F_P} (@x.\langle f\rangle nil) \wedge \bigwedge_{v \in V_P[z/x]} (@v.[(f_1 + \ldots + f_k)^*]\neg x)$$

Then $H[x := H(z)] \models \Phi$ by the substitution lemma, so by assumption

$$H[x := H(z)][x := n][\bar{f} := \bar{\rho}] \models \Psi$$

hold for any $n \in \mathbb{N} \setminus \mathcal{R}_{H[x:=H(z)]}(V_P, F_P)$, and for any $\bar{\rho}$ such that for all i:

$$\begin{aligned}\rho_i &= H[x := H(z)](f_i)[n := (H[x := H(z)](nil))] \\ &= H(f_i)[n := H(nil)]\end{aligned}$$

Now note that since $H \models \bigwedge_{v \in V_P[z/x]} @v.[(f_1 + \ldots + f_k)^*]\neg x$, we have that $H(x) \notin \mathcal{R}_H(V_P[z/x], F_P) = \mathcal{R}_{H[x:=H(z)]}(V_P, F_P)$. Thus we can take $n = H(x)$:

$$H[x := H(z)][x := H(x)][\bar{f} := \bar{\rho}] \models \Psi$$

The first substitution has no effect on the second substitution $x := H(x)$, and so

$$H[x := H(z)][x := H(x)][\bar{f} := \bar{\rho}] = H[\bar{f} := \bar{\rho}]$$

But now $H[\bar{f} := \bar{\rho}] \models \Psi$, and since $H \models \bigwedge_{f \in F_P}(@x.\langle f \rangle nil)$, we have for each $f \in F_P$: $H(f)(H(x)) = H(nil)$, and consequently $H(f) = H(f)[H(x) := H(nil)]$, so $H[\bar{f} := \bar{\rho}] = H$. We conclude $H \models \Psi$. □

□

Appendix B

Maude source code

tablo.maude

```
mod TABLO is
inc NAT  .
inc BOOL .
inc DL .
including FUNCTIONS .
sorts Config DelimConfig .
subsort Model < Config .
op i : -> Field .
op empty : -> Config .
op (_|_)  : NewWorlds Model -> Config [left id: empty].
op _,_ : Config Config  -> Config  [assoc id: empty] .
op {_} : Config -> DelimConfig .
vars p q r s : Form .
var lit : Literal .
var f g : Field .
vars  k l m n : Nat .
var N : Next .
var R : NewWorlds .
var M : Model .
var U T : Transitions .
var W : Worlds .
var x y z : Nominal .
var xl : Nominals .
var P Q : VarForms .
```

```
vars a b c : Prog .

var C : Config .

eq < f , n , m > , < f , n , m > = < f , n , m > .

*** Input methods: init(p) searches for a model of p where p is a formula in dynamic logic,
*** P |= Q checks entailment of rooted formulas P and Q

op _|=_ : VarForms VarForms -> DelimConfig .

op i(_;_) : VarForms VarForms -> Config .

op initmodel : VarForms -> NewWorlds .

op init : Form -> DelimConfig .

eq P |= Q = {i(P ; Q)} .

eq i(P ; empty) = empty .

eq i(P ; (x . p) , Q) = (initmodel((x . (- p)) , P) |  (empty ; empty ; 0)) , i(P ; Q) .

eq initmodel(empty) = empty .

eq initmodel((x . p) , Q) = [0, i, x & p, true, none] , initmodel(Q) .

eq init(p) = {[0, i , p , true , none] | (empty ; empty ; 0)} .


*** Tableaux construction

rl [rminitlink] : < i , n , m > , T => T .

eq ([n , f, false & p , r , N], R | M) = empty .
eq ([n , f, p , false & r , N], R | M) = empty .
eq {(R | (((n . false & p), W) ; T ; m)) , C } = { C } .

rl [diamond] : {([n , f , (< g > p) & q, r , N], R | M) , C} =>
{([n , f , q ,  r & < g > p,  add(g, p, N)], R | M) , C} .

rl [box] : {([n , f , ([ g ] p) & q, r , N], R | M) , C} =>
{([n , f , q ,  r & [ g ] p, add(g, p, N)], R | M) , C} .

crl [literal] : {([n , f, lit & p , r , N], R | M) , C} =>
{([n , f, p , r & lit , N], R | M) , C} if lit =/= true .

rl [newworld1] : {([ n , f , true,  p , N] , R | (((k . p) , W) ; T ; m)) , C} =>
{(R | (((k . p) , W) ;  (< f , n, k > , T) ; m)) , C} .

crl [newworld2] : {([ n , f , true,  p , N] , R | (W ; T ; m)) , C} =>
 {(new(m, N), R | (((m . p), W) ; (< f , n, m > , T) ; m + 1)) , C}
 if in(p,W) =/= true .

rl [nominal] : {(empty | (((n . x & p), (k . x & q), W) ; T ; m)) , C}  =>
 {(empty | (((min(n , k) . x & p & q), W) ; subst(T , max(n , k), min(n , k)); m)) , C} .

crl [fieldid] :
{( empty | (((k . p), (l . q), W) ; (< f , n , k >, < f , n , l >, T); m)) , C}
=>
{( empty | (((min(k , l) . p & q),  W) ; < f , n , min(k , l) > ,
subst(T , max(k , l), min(k , l)) ; m)) , C } if f =/= i .

rl [split1] :
{([n , f , (p | q) & s , r ,  N], R | M) , C} =>
{([n , f , p & s, r & (p | q),  N], R | M) , ([n , f , q & s, r & (p | q),  N], R | M) , C} .

*** this rule is necessary for things like < x * > y .
crl [already-done] :
 {([n , f , p & q , p & s , N] , R | M) , C } =>
 {([n , f , q , p & s , N] , R | M) , C} if p =/= true .

rl [split2] :
{([n , f , (< a * > p ) & s , r ,  N], R | M) , C} =>
{([n , f , p & s , r & < a * > p,  N], R | M) , ([n , f , (< a > < a * > p ) & s , r & < a * > p ,
N], R | M) , C} .
```

```
*** in the first world we write r instead of < a *> p, which makes sure that in this world
*** the star check is skipped - which is a small optimization

rl [boxstar] :
 {([n , f , ([ a * ] p ) & s , r , N], R | M) , C} =>
 {([n , f , p & s & [ a ] [a * ] p, r & ([ a * ] p) , N], R | M) , C} .

crl [starcheck-diamond] :
{ ((( n . q & < a * > p) , W) ; T ; m) , C } => { C }
if stable (W) and stable (T) and not Reach(m x a , (n . q & < a * > p) , W , T , n , p) .


endm
```

verify.maude

```
mod VERIFY is


including TABLO .

including SUBST .

sorts Expr Stat Method Methods MethodName AConfig AConfigs PConfig Fields VConditions .

subsort Field < Fields .

subsort DelimConfig < VConditions .

subsort Nominal < Expr .

subsort AConfig < AConfigs .

subsort Method < Methods .

subsort MethodName < Stat .

op _..._ : Nominal Field -> Expr .

op _=_ : Expr Expr -> Bool .

op new : -> Expr .

op E : -> Stat .

op empty : -> AConfig .

op none : -> Methods .

op none : -> Fields .

op success : -> PConfig .

op none : -> VConditions .

op _,_ : AConfigs AConfigs -> AConfigs [assoc right id: empty] .

op _,_ : VConditions VConditions -> VConditions [assoc id: none].

op _,_ : Methods Methods -> Methods [assoc comm right id: none] .

op progvars : Stat -> Nominals .

op _:_ : Fields Fields -> Fields [assoc comm id: none] .

op _[_|_] : Stat Fields Nominals -> Stat .

op fields-nil : Fields -> Form .

op unreachable : Nominal Nominals Fields -> VarForms .

op _:=_ : Expr Expr -> Stat  .

op _;_ : Stat Stat -> Stat [assoc right id: E] .
```

```
op _while_do_od : VarForms VarForms Stat -> Stat .

op if_then_else_fi : VarForms Stat Stat -> Stat .

op (_._._._._) : MethodName VarForms Stat VarForms -> Method .

op <_._._._._._> : MethodName VarForms Stat VarForms Bool -> Method .

op <_,_,_> : VarForms Stat VarForms -> AConfig .

op <_,_,_> : AConfigs Methods VConditions -> PConfig .

op done : Methods -> Bool .

op _' : Nominal -> Nominal .

op _' : Field -> Field .

op ver : Methods -> PConfig .

op plus : Fields -> Prog .

vars S S1 S2 : Stat .

var e, e2 : Expr .

vars p q r s : Form .

vars P Q R O : VarForms .

var V : Nominals .

var VC : VConditions .

vars x y z head tmp : Nominal .

var f g next : Field .

var F : Fields .

var C : AConfigs .

var m : Method .

var b : Bool .

var M : Methods .

var mn m1 m2 : MethodName .

eq done ( none ) = true .
eq done ( < mn . P . S . Q . false > , M ) = false .
eq done ( < mn . P . S . Q . true > , M ) = done ( M ) .

eq ver(M) = < empty , M , none > .

eq fields-nil(none) = true .
eq fields-nil(f) = < f > nil .
eq fields-nil(f : F) = fields-nil(F) & < f > nil .

eq plus(empty) = true .
eq plus(f) = f .
eq plus(f : F) = f + plus(F) .

eq unreachable(x , empty , F) = empty .
eq unreachable(x , x V , F) = unreachable(x , V , F) .
ceq unreachable(x , z V , F) = unreachable(x , V , F) , (z . [(plus(F)) *](- x)) if x =/= z .

eq ( mn . P . S . Q ) = < mn . P . S . Q . false > .

rl [start] : < C , (< mn . P . S . Q . false > , M), VC > =>
 < (< P , S , Q > , C) , (< mn . P . S . Q . true > , M), VC > .

rl [method-termination] : < (< P , E , Q > , C) , M , VC > =>
 < C , M , (P |= Q) , VC > .
```

```
rl [ifthenelse] : < P , (if Q then S1 else S2 fi) ; S, R > =>
 < P , Q , S1 ; S , R > , < P , (- Q) , S2 ; S , R > .

*** check : P |= Q ,
rl [while] : < (< P , (Q while R do S1 od) ; S , O > , C) , M , VC > =>
 < (< Q , R , S1 , Q > , < Q , (- R) , S , O > , C) , M , (P |= Q) , VC > .

*** check : P |= R
rl [proc] : < (< P , mn ; S , Q > , C) , (< mn . R . S1 . O . b > , M) , VC > =>
 < (< O , S , Q > , C ) , < mn . R . S1 . O . b > , M , (P |= R) , VC > .

rl [asss0] : < P , (x := x) ; S , Q > =>
 < P , S , Q > .

crl [ass1] : < P , (x := y) ; S , Q > =>
 < (P [ x := x ' ]) , (x . y), S , Q >  if x =/= y .

rl [ass2] : < P , (x := (y .. f)) ; S , Q > =>
 < (P [ x := x ' ]) , (y [ x := x '] . (< f > x)) , S , Q > .

rl [ass3] : < P , ((x .. f) := y) ; S , Q > =>
 < (P [ f := ((x ; (f ')) + ((- x) ; f)) ]) , (x . (- nil) & (< f > y)) , S , Q > .

rl [obj-create] : < P , (x := new [ F | V ] ) ; S , Q > =>
 < (P [ x := x ' ]) , (x . fields-nil(F)) , unreachable(x , V , F), S , Q > .

rl [cond-done] : < C , M , { empty } , VC > =>
 < C , M , VC > .

crl [termination] : < empty , M , none > => success if done( M ).

endm
```

dl.maude

```
*** This module contains the syntactic definition dynamic logic
*** with fields as basic programs, together with some reduction axioms.

mod DL is

sorts Field Nominal Literal Prog Form VarForms .

subsort Nominal < Literal .

subsort Field < Prog .

subsort Literal < Prog .

subsort Literal < Form .

op true : -> Literal .

op false : -> Literal .

op nil : -> Nominal .

op empty : -> VarForms .

op _._ : Nominal Form -> VarForms .
op _,_ : VarForms VarForms -> VarForms [assoc id: empty].

*** Syntax
op -_ : Literal -> Literal .

op -_ : Form -> Form .

op -_ : VarForms -> VarForms .

op _+_ : Prog Prog -> Prog .

op _;_ : Prog Prog -> Prog .

op _* : Prog -> Prog .
```

```
op _ : Literal -> Prog .

op _&_ : Form Form -> Form [assoc comm id: true ] .

op _|_ : Form Form -> Form [assoc comm] .

op <_>_ : Prog Form -> Form .

op [_]_ : Prog Form -> Form .

op _' : Nominal -> Nominal .

op _' : Field -> Field .

vars p q r s : Form .

var lit : Literal .

var x y : Nominal .

var f g : Field .

var a b : Prog .

var P : VarForms .

eq (x . p), (x . q) = x . (p & q) .

*** Equations for reduction of formulas
eq - (- p) = p .
eq p & (- p) = false .
eq p | (- p) = true .

*** ceq false & p = false if (p =/= true) .

eq (p & p) = p .

*** test:
eq < lit > p = lit & p .
eq [ lit ] p = - lit | p .


*** negation:
eq - empty = empty .
eq - ( (x . p) , P ) = (x . (- p)) , (- P) .
eq - true = false .
eq - false = true .
ceq - (p & q) = (- p) | (- q) if (p =/= true) and (q =/= true) .
eq - (p | q) = (- p) & (- q) .
eq - (< a > p) = [ a ] - p .
eq - ([ a ] p) = < a > - p .

*** box and diamond
eq [a + b] p = ([a] p) & ([b] p) .
eq < a + b > p = (< a > p) | (< b > p) .
eq [a ; b] p = [a][b] p .
eq < a ; b > p = < a >< b >  p .


endm
```

functions.maude

```
mod FUNCTIONS is

inc NAT .

inc DL .


sorts Nominals World Worlds Transitions
Next Model NewWorlds .

subsort Nominal < Nominals .
```

```
subsort World < Worlds .

op none : -> Next .

op empty : -> Nominals .

op empty : -> NewWorlds .

op empty : -> Worlds .

op empty : -> Transitions .

op _._ : Nat Form -> World .

op _,_ : Worlds Worlds  -> Worlds  [assoc comm id: empty] .

op _ _ : Nominals Nominals -> Nominals [idem assoc comm id: empty] .

op <_,_,_> : Field Nat Nat -> Transitions .

op <_,_> : Nat Nat -> Transitions .

op _,_ : Transitions Transitions -> Transitions [assoc idem comm id: empty] .

op (_;_;_) : Worlds Transitions Nat -> Model .

op (_,_) : Field Form  -> Next .

op _,_ : Next Next -> Next [assoc comm id: none] .

*** New worlds: worlds which are being processed; if [n, f, p, q, N] is a newworld then there already
*** is a world with id n, from which there -will- be an f-link to this new world, when its created. p
*** is the formula that still needs to be made true, and q is already 'done'. N is a set of worlds
*** which will be newworlds as soon as this world is really constructed; these are the outgoing fields
*** of the new world.
op [_,_,_,_,_] : Nat Field Form Form Next -> NewWorlds .

op _,_ : NewWorlds NewWorlds -> NewWorlds [assoc id: empty].

op new : Nat Next -> NewWorlds .

op add : Field Form Next -> Next .

op in : Form Worlds -> Bool .

op vars : Form -> Nominals .

op vars : VarForms -> Nominals .

op subst : Nat Nat Nat -> Nat .

op subst : Transitions Nat Nat -> Transitions .

op subst-nw : NewWorlds Nat Nat -> NewWorlds .

op Sem : Prog Worlds Transitions -> Transitions .

op _;_ : Transitions Transitions -> Transitions .

op _x_ : Nat Prog -> Prog .

op Reach : Prog Worlds Transitions Nat  Form -> Bool .

op Reach : Transitions Nat Worlds Form -> Bool .

op find : Worlds Form -> Worlds .

op stable : Worlds -> Bool .

op stable : Transitions -> Bool .

vars p q r s : Form .

var lit : Literal .
```

```
var f g : Field .

vars  k l m n : Nat .

var N : Next .

var R : NewWorlds .

var M : Model .

var U T : Transitions .

var W : Worlds .

var x y z : Nominal .

vars a b c : Prog .

var P : VarForms .

*** Definition of vars

eq vars((empty).VarForms) = empty .
eq vars((x . p) , P) = x vars(p) vars(P) .

eq vars (x) = x .
eq vars (p | q) = vars(p) vars(q) .
ceq vars (p & q) = (vars(p) vars(q)) if (p =/= true) and (q =/= true) .
eq vars (- p) = vars(p) .
eq vars (< a * > p) = vars ( < a > p) .
eq vars ([ a * ] p) = vars ( [ a ] p) .
eq vars (< f > p) = vars ( p ) .
eq vars ([ f ] p) = vars ( p ) .
eq vars (true) = empty .
eq vars (false) = empty .
eq x x = x . *** somehow the idem keyword is not enough

*** Helper functions

eq < k , l > , < k , l > , T = < k , l > , T .

eq Sem(f , W , < f , n , k >, T) = < n , k > , Sem(f , W , T) .

eq Sem(f , W ,  T) = empty [owise] .

eq Sem(lit , (n . lit & p) , W , T) = < n , n > , Sem(lit , W , T) .

eq Sem(lit , W , T) = empty [owise] .

eq Sem(a + b , W , T) = Sem(a , W , T) , Sem(b , W , T) .

eq Sem(a ; b , W , T) = Sem(a , W , T) ; Sem(b , W , T) .

eq (empty ; U) = empty  .

ceq (< k , l >  , T) ; U = (< k , l > ; U ) , (T ; U) if T =/= empty .

eq < k , l > ; (< l , m > , U) = < k , m > , (< k , l > ; U) .

eq < k , l > ; U = empty [owise] .

eq Sem(0 x a , W , T) = Sem(a , W , T).

ceq Sem(k x a , W , T) =
Sem(sd(k,1) x a , W , T) , (Sem(a , W , T) ;  Sem(sd(k,1) x a , W , T)) if k > 0 .

eq Reach( a , W , T , n , p) = Reach(Sem(a , W , T), n, W , p) .

eq Reach(T , n , (n . p & q) , W , p) = true .
eq Reach((< n , m > , T ), n , (m . p & q) , W , p) = true .

eq Reach((< n , m > , T ), n , W , p) = false [owise] .

eq new(n, none) = empty .

eq new(n, (f,p) , N) = [n , f , p , true , none] , new(n , N) .
```

```
eq add(f, p, (f, q), N) = (f, p & q), N .

eq add(f, p, N) = (f, p), N [owise] .

eq in(p,empty) = false .

eq in(p, (n . p), W) = true .

ceq in(p, (n . q), W) = in(p, W) if p =/= q .

ceq subst(k , l , m) = m if k = l .

ceq subst(k, l , m) = k if k =/= l .

eq subst(empty , n , m) = empty .

eq subst( < f, k , l > , T , n , m) =
< f , subst(k , n , m) , subst(l , n , m) > , subst(T , n , m) .

eq subst-nw( empty , n , m ) = empty .

eq subst-nw( [n , f , p , q , N] , R , k , l) =
[ subst(n , k , l) , f , p , q , N] , subst-nw(R , k , l) .

eq stable((n . x & p) , (m . x & q) , W) = false .

eq stable(W) = true [owise] .

ceq stable(< f , n , m > , < f , n , l > , T) = false if m =/= l .

eq stable(T) = true [owise].

endm
```

subst.maude

```
*** This module supplies a substitution of nominals for nominals,
*** and substitution of fields for general navigation expressions.
mod SUBST is

including DL .

sorts Lhs Rhs .

subsort Nominal < Lhs .
subsort Field < Lhs .

subsort Nominal < Rhs .
subsort Prog < Rhs .

vars p q : Form .

vars x y z self : Nominal .

var l : Lhs .

var r : Rhs .

var f g : Field .

vars a b : Prog .

var P : VarForms .

*** the substitution operators
op _[_:=_] : Form Nominal Nominal -> Form .
op _[_:=_] : Prog Field Prog -> Prog .

op _[_:=_] : VarForms Lhs Rhs -> VarForms .

eq empty [ l := r ] = empty .
eq ((x . p) , P) [ l := r ] = ( ( x [ l := r] . (p [ l := r]) ) , (P [ l := r]) ) .

*** the prime operator needs support, too (is in DL)
```

```
*** op _' : Nominal -> Nominal .
*** op _' : Field -> Field .

*** substitution of nominals - base cases
eq x [ x := y ] = y .
eq z [ x := y ] = z [owise] .
eq f [ x := y ] = f .

*** substitution of fields - base cases
eq x [ f := a ] = x .
eq f [ f := a ] = a .
eq g [ f := a ] = g [owise] .

*** inductive definition of the other cases, which are the same for both
*** substitution of nominals and of fields
eq true [ l := r ] = true .

eq false [ l := r ] = false .

eq (z ') [ l := r ] = (z [ l := r ]) ' .

eq (- p) [ l := r ] = - (p [ l := r ]) .

ceq (p & q) [ l := r ] = (p [ l := r ]) & (q [ l := r ])
   if (p =/= true and q =/= true) .

eq (p | q) [ l := r ] = (p [ l := r ]) | (q [ l := r ]) .

eq (< a > p ) [ l := r ] = < ( a [ l := r ] ) > ( p [ l := r ] ) .

eq ([ a ] p ) [ l := r ] = [ ( a [ l := r ] ) ] ( p [ l := r ] ) .

eq (a ; b) [ l := r ] = ( (a [ l := r ]) ; (b [ l := r ]) ) .

eq (a *) [ l := r ] = ( (a [ l := r ] ) * ) .

eq (a + b) [ l := r ] = ((a [ l := r ] ) + (b [ l := r ])) .

endm
```