Sequential, Parallel, and Quantified Updates of First-Order Structures

Philipp Rümmer

Department of Computer Science and Engineering, Chalmers University of Technology and Göteborg University, SE-412 96 Göteborg, Sweden
philipp@cs.chalmers.se

Abstract. We present a datastructure for storing memory contents of imperative programs during symbolic execution—a technique frequently used for program verification and testing. The concept, called updates, can be integrated in dynamic logic as runtime infrastructure and models both stack and heap. Here, updates are systematically developed as an imperative programming language that provides the following constructs: assignments, guards, sequential composition and bounded as well as unbounded parallel composition. The language is equipped both with a denotational semantics and a correct rewriting system for execution, whereby the latter is a generalisation of the syntactic application of substitutions. The normalisation of updates is discussed. The complete theory of updates has been formalised using Isabelle/HOL.

1 Introduction

First-Order Dynamic Logic [1] is a program logic that enables to reason about the relation between the initial and final states of imperative programs. One way to build calculi for dynamic logic is to follow the symbolic execution paradigm and to execute programs (symbolically) in forward direction. This requires infrastructure for storing the memory contents of the program, for updating the contents when assignments occur and for accessing information whenever the program reads from memory. Sequent calculi for dynamic logic often represent memory using formulas and handle state changes by renaming variables and by relating pre- and post-states with equations. All information about the considered program states is stored in the side-formulas Γ, Δ of a sequent $\Gamma \vdash \langle \alpha \rangle \phi, \Delta$, like in inequations $0 \dot{<} x$ and equations $x' \doteq x + 1$.

As an alternative, this paper presents a datastructure called *Updates*, which are a generalisation of substitutions designed for storing symbolic memory contents. When using updates, typical sequents during symbolic execution have the shape $\Gamma \vdash \{u\} \langle \alpha \rangle \phi, \Delta$. The program α is preceded by an update u that determines parts of the program state, for instance the update $x := x + 1$. Compared with side-formulas, updates (i) attach information about the program state directly to the program, (ii) avoid the introduction of new symbols, (iii) can be simplified and avoid the storage of obsolete information, like of assignments that have been overridden by other assignments, (iv) represent accesses to variables,

array cells or instance attributes (in object-oriented languages) in a uniform way, (v) delay case-distinctions that can become necessary due to aliasing, (vi) can be eliminated mechanically once a program has been worked off completely.

Historically, updates have evolved over years as a central component of the KeY system [2], a system for deductive verification of Java programs. They are used both for interactive and automated verification. In the present paper, we define updates as a formal language (independently of particular program logics) and give them a denotational semantics based on model-theoretic semantics of first-order predicate logic. The language is proposed as an intermediate language to which sequential parts of more complicated languages (like Java) can stepwise be translated. The thesis [3] related to this paper gives a rewriting system that allows to execute or eliminate updates mechanically. The main contributions of the paper are new update constructs (in particular quantification), the development of a complete metatheory of updates and its formalisation[1] using the Isabelle/HOL proof assistant [4], including proofs of all lemmas about updates that are given in the present paper or in [3].

The paper is organised as follows: Sect. 2 motivates updates through an example. Sect. 3 and 4 introduce syntax and semantics of a basic version of updates in the context of a minimalist first-order logic. Sect. 5 describes the rewriting system for executing updates. Sect. 6 adds an operator for sequential composition to the update language. Sect. 7 shows how heap structures can be modelled and modified using updates, which is applied in Sect. 8 about symbolic execution. Sect. 9 discusses laws for simplification of updates.

2 Updates for Symbolic Execution in Dynamic Logic

We give an example for symbolic execution using updates in dynamic logic. Notation and constructs used here are later introduced in detail. The program fragment *max* is written in a Java-like language and is executed in the context of a class/record *List* representing doubly-linked lists with attributes *next*, *prev* and *val* for the successor, predecessor and value of list nodes:

$$max \quad = \quad \texttt{if}\ (a.val \dot{<} a.next.val)\ g = a.next.val; \texttt{else}\ g = a.val;$$

where a and g are program variables pointing to list nodes. The initial state of program execution is specified in an imperative way using an update:

$$\begin{aligned} init \quad = \quad & a.prev := nil \mid b.next := nil \mid a.next := b \mid b.prev := a \mid \\ & a.val := c \mid b.val := d \end{aligned}$$

init can be read as a program that is executing a number of assignments in parallel and that is setting up a list with nodes a and b. In case $a \doteq b$—which is possible because we do not specify the opposite—the two nodes will collapse to the single node of a cyclic list and will carry value d: assignments that literally

[1] `www.cs.chalmers.se/~philipp/updates.thy`, $\approx$ 3500 lines Isabelle/Isar code

occur later ($b.val := d$) can override earlier assignments ($a.val := c$). This means that parallel composition in updates also has a sequential component: while the left- and right-hand sides of the assignments are all evaluated in parallel, the actual writing to locations is carried out sequentially from left to right.

When adding updates to a dynamic logic, they can be placed in front of modal operators for programs, like in $\{init\}\ \langle\, max \,\rangle\ \phi$. The diamond formula $\langle\, max \,\rangle\ \phi$ alone expresses that a given formula ϕ holds in at least one final state of max. Putting the update $init$ in front means that first $init$ and then the program max is supposed to be executed—$init$ sets up the pre-state of max.

We execute max symbolically by working off the statements in forward direction. Effects of the program are either appended to the update $init$ or are translated to first-order connectives. We denote execution steps of max by $\rightsquigarrow$ and write $\equiv$ for an update simplification step. $init$ is used as an abbreviation.

$$\{init\}\ \langle\, \texttt{if}\ (a.val \doteq a.next.val)\ g = a.next.val; \texttt{else}\ g = a.val; \rangle\ \phi$$

A conditional statement can be translated to propositional connectives. The branch condition is $co = (a.val \doteq a.next.val)$.

$$\rightsquigarrow \quad \{init\}\ \big((co \wedge \langle\, g = a.next.val; \rangle\ \phi) \vee (\neg co \wedge \langle\, g = a.val; \rangle\ \phi)\big)$$

The application of $init$ distributes through propositional connectives. Applying $init$ to co yields the condition $co' = (\{init\}\ co) \equiv ((\text{if}\ \ a \doteq b\ \ \text{then}\ \ d\ \ \text{else}\ \ c) \doteq d)$.

$$\equiv \quad (co' \wedge \{init\}\ \langle\, g = a.next.val; \rangle\ \phi \vee (\neg co' \wedge \{init\}\ \langle\, g = a.val; \rangle\ \phi)$$

The program assignments are turned into update assignments that are sequentially (;) connected with $init$.

$$\rightsquigarrow \quad (co' \wedge \{init\,;\ g := a.next.val\}\ \phi) \vee (\neg co' \wedge \{init\,;\ g := a.val\}\ \phi)$$

The updates are simplified by turning sequential composition ; into parallel composition | . The update $init$ has to be applied to the right-hand sides, which become $(\{init\}\ a.next.val) \equiv d$ and $(\{init\}\ a.val) \equiv (\text{if}\ \ a \doteq b\ \ \text{then}\ \ d\ \ \text{else}\ \ c)$.

$$\equiv \quad (co' \wedge \{init \mid g := d\}\ \phi) \vee (\neg co' \wedge \{init \mid g := (\text{if}\ \ a \doteq b\ \ \text{then}\ \ d\ \ \text{else}\ \ c)\}\ \phi)$$

The last formula is logically equivalent to the original formula $\{init\}\ \langle\, max \,\rangle\ \phi$ and can further be simplified by applying the updates to ϕ. An implementation like in KeY can, of course, easily carry out all shown steps automatically.

3 Syntax of Terms, Formulas, and Updates

The present paper is a self-contained account on updates. To this end, we abstract from concrete program logics and define syntax and semantics of a (minimalist)[2] first-order logic that is equipped with updates. Updates can, however, be integrated in virtually any predicate logic, e.g., in dynamic logic.

We first define a basic version of our logic that contains the most common constructors for terms and formulas (see e.g. [5]), the equality predicate $\doteq$ and

[2] We do not include many common features like arbitrary predicate symbols, in order to keep the presentation concise. Adding such concepts is straightforward.

a strict order relation $\dot{<}$, as well as operators for minimum and conditional terms. The two latter are not strictly necessary, but enable a simpler definition of laws and rewriting rules. In this section, updates are only equipped with the connectives for parallelism, guards and quantification, sequential composition is added later in Sect. 6.

In order to define the syntax of the logic, we need (i) a vocabulary (Σ, α) of function symbols, where $\alpha : \Sigma \to \mathbb{N}$ defines the arity of each symbol, and (ii) an infinite set *Var* of variables.

Definition 1. *The sets Ter, For and Upd of terms, formulas and updates are defined by the following grammar, in which $x \in Var$ ranges over variables and $f \in \Sigma$ over functions:*

$$\begin{aligned}
Ter &::= x \mid f(Ter, \ldots, Ter) \mid \text{if } For \text{ then } Ter \text{ else } Ter \mid \min x.\, For \mid \{Upd\}\ Ter \\
For &::= true \mid false \mid For \wedge For \mid For \vee For \mid \neg For \mid \forall x.\, For \mid \exists x.\, For \mid \\
&\quad\ Ter \doteq Ter \mid Ter \dot{<} Ter \mid \{Upd\}\ For \\
Upd &::= \texttt{skip} \mid f(Ter, \ldots, Ter) := Ter \mid Upd \,|\, Upd \mid \texttt{if } For\ \{Upd\} \mid \texttt{for } x\ \{Upd\}
\end{aligned}$$

The update constructors represent the empty update `skip`, assignments to function terms $f(s_1, \ldots, s_n) := t$, parallel updates $u_1 \mid u_2$, guarded updates $\texttt{if } \phi\ \{u\}$, and quantified updates $\texttt{for } x\ \{u\}$. The possibility of having function terms as left-hand sides of assignments is crucial for modelling heaps. In Sect. 2, expressions like *a.prev* are really function terms $prev(a)$, but we use the more common notation from programming languages. More details are given in Sect. 7. There are also constructors for applying updates to terms and to formulas (like $\{u\}\ \phi$).

We mostly use vector notation for the arguments $\bar{t}$ of functions. Operations on terms are extended canonically or in an obvious way to vectors, for instance $f(\{u\}\ \bar{t}) = f(\{u\}\ t_1, \ldots, \{u\}\ t_n)$, $\mathrm{val}_{S,\beta}(\bar{t}) = (\mathrm{val}_{S,\beta}(t_1), \ldots, \mathrm{val}_{S,\beta}(t_n))$.

4 Semantics of Terms, Formulas, and Updates

The meaning of terms and formulas is defined using classical model-theoretic semantics. We consider interpretations as mappings from *locations* to *individuals* of a universe U (the predicates $\doteq$ and $\dot{<}$ are handled separately):

Definition 2. *Given a vocabulary (Σ, α) of function symbols and an arbitrary set U, we define the set $Loc_{(\Sigma,\alpha),U}$ of* locations *over (Σ, α) and U by*

$$Loc_{(\Sigma,\alpha),U} := \{\langle f, (a_1, \ldots, a_n)\rangle \mid f \in \Sigma,\ \alpha(f) = n,\ a_1, \ldots, a_n \in U\}\ .$$

If the indexes are clear from the context, we just write Loc instead of $Loc_{(\Sigma,\alpha),U}$.

The following definition of structures/algebras deviates from common definitions in the addition of a strict well-ordering on the universe.[3] The well-ordering is used for resolving clashes that can occur in quantified updates (see Example 1 and Sect. 8).

[3] As every set can be well-ordered (based on Zermelo-Fraenkel set theory [6]) this does not restrict the range of considered universes. Because the well-ordering is also

Definition 3. *Suppose that a vocabulary (Σ, α) of function symbols is given. A* well-ordered algebra *over (Σ, α) is a tuple $S = (U, <, I)$, where*

- *U is an arbitrary non-empty set (the* universe*),*
- *$<$ is a strict well-ordering on U, i.e., a binary relation with the properties*[4]
  - *Irreflexivity: $a \not< a$ for all $a \in U$,*
  - *Transitivity: $a_1 < a_2$, $a_2 < a_3$ entails $a_1 < a_3$ $(a_1, a_2, a_3 \in U)$,*
  - *Well-orderedness: Every non-empty set $A \subseteq U$ contains a least element $\min_< A \in A$ such that $\min_< A < a$ for all $a \in A \backslash \{\min_< A\}$,*
- *I is a (total) mapping $Loc_{(\Sigma,\alpha),U} \to U$ (the* interpretation*).*

A partial interpretation *is a partial function $Loc_{(\Sigma,\alpha),U} \nrightarrow U$.*

A (partial) function $f : M \nrightarrow N$ is here considered as a subset of the cartesian product $M \times N$. For combining and modifying interpretations, we frequently make use of the *overriding* operator $\oplus$, which can be found in Z [7] and many other specification languages. For two (partial or total) functions $f, g : M \nrightarrow N$ we define

$$f \oplus g \;:=\; \{(a \mapsto b) \in f \mid \text{for all } c\text{: } (a \mapsto c) \notin g\} \cup g \;,$$

i.e., g overrides f but leaves f unchanged at points where g is not defined. For $S = (U, <, I)$, we also write $S \oplus A := (U, <, I \oplus A)$ as a shorthand notation.

Definition 4. *A variable assignment over a set Var of variables and a well-ordered algebra $(U, <, I)$ is a mapping $\beta : Var \to U$.*

Given a variable assignment β, we denote the assignment that is altered in exactly one point as is common:

$$\beta^a_x(y) \;:=\; \begin{cases} a & \text{for } x = y \\ \beta(y) & \text{otherwise} \end{cases}$$

From now on, we consider the vocabulary (Σ, α) and *Var* as fixed.

Definition 5. *Given a well-ordered algebra $S = (U, <, I)$ and a variable assignment β, we define the* evaluation *of terms, formulas and updates through the equations of Table 1 as the (overloaded) mapping*

$$\mathrm{val}_{S,\beta} : Ter \to U, \quad \mathrm{val}_{S,\beta} : For \to \{tt, ff\}, \quad \mathrm{val}_{S,\beta} : Upd \to (Loc \nrightarrow U),$$

i.e., in particular updates are evaluated to partial interpretations.

---

accessible through the predicate $\dot{<}$, however, the expressiveness of the logic goes beyond pure first-order predicate logic. One can, for instance, axiomatise natural numbers up to isomorphism with a finite set of formulas. In our experience, this is not a problem for the application of updates, because quantification in updates will in practice only be used for variables representing integers, objects or similar types. On such domains, appropriate well-orderings are readily available and have to be handled anyway.

[4] Note, that well-orderings are linear, i.e., $a < b$, $a = b$, or $b < a$ for arbitrary $a, b \in U$. Further, well-orderings are well-founded—there are no infinite descending chains—which enables us to use well-founded recursion when defining update evaluation.

Table 1. Evaluation of Terms, Formulas, and Updates

For terms:

$$\begin{aligned}
\mathrm{val}_{S,\beta}(x) &= \beta(x) && (x \in \mathit{Var})\\
\mathrm{val}_{S,\beta}(f(\bar{t})) &= I\langle f, \mathrm{val}_{S,\beta}(\bar{t})\rangle && (S = (U, <, I))\\
\mathrm{val}_{S,\beta}(\text{if } \phi \text{ then } t_1 \text{ else } t_2) &= \begin{cases} \mathrm{val}_{S,\beta}(t_1) & \text{for } \mathrm{val}_{S,\beta}(\phi) = tt \\ \mathrm{val}_{S,\beta}(t_2) & \text{otherwise} \end{cases}\\
\mathrm{val}_{S,\beta}(\min x.\, \phi) &= \begin{cases} \min_< A & \text{for } A \neq \emptyset \\ \min_< U & \text{otherwise} \end{cases}
\end{aligned}$$

where $S = (U, <, I)$ and $A = \{a \in U \mid \mathrm{val}_{S,\beta^a_x}(\phi) = tt\}$

For formulas:

$$\begin{aligned}
\mathrm{val}_{S,\beta}(\mathit{true}) &= tt, \qquad \mathrm{val}_{S,\beta}(\mathit{false}) = ff\\
\mathrm{val}_{S,\beta}(\phi_1 \wedge \phi_2) &= tt \quad \text{iff} \quad ff \notin \{\mathrm{val}_{S,\beta}(\phi_1), \mathrm{val}_{S,\beta}(\phi_2)\}\\
\mathrm{val}_{S,\beta}(\phi_1 \vee \phi_2) &= tt \quad \text{iff} \quad tt \in \{\mathrm{val}_{S,\beta}(\phi_1), \mathrm{val}_{S,\beta}(\phi_2)\}\\
\mathrm{val}_{S,\beta}(\neg\phi) &= tt \quad \text{iff} \quad \mathrm{val}_{S,\beta}(\phi) = ff\\
\mathrm{val}_{S,\beta}(\forall x.\, \phi) &= tt \quad \text{iff} \quad ff \notin \{\mathrm{val}_{S,\beta^a_x}(\phi) \mid a \in U\}\\
\mathrm{val}_{S,\beta}(\exists x.\, \phi) &= tt \quad \text{iff} \quad tt \in \{\mathrm{val}_{S,\beta^a_x}(\phi) \mid a \in U\}\\
\mathrm{val}_{S,\beta}(t_1 \doteq t_2) &= tt \quad \text{iff} \quad \mathrm{val}_{S,\beta}(t_1) = \mathrm{val}_{S,\beta}(t_2)\\
\mathrm{val}_{S,\beta}(t_1 \dot{<} t_2) &= tt \quad \text{iff} \quad \mathrm{val}_{S,\beta}(t_1) < \mathrm{val}_{S,\beta}(t_2) && (S = (U, <, I))
\end{aligned}$$

For updates:

$$\begin{aligned}
\mathrm{val}_{S,\beta}(\texttt{skip}) &= \emptyset\\
\mathrm{val}_{S,\beta}(f(\bar{s}) := t) &= \{\langle f, \mathrm{val}_{S,\beta}(\bar{s})\rangle \mapsto \mathrm{val}_{S,\beta}(t)\}\\
\mathrm{val}_{S,\beta}(u_1 \mid u_2) &= \mathrm{val}_{S,\beta}(u_1) \oplus \mathrm{val}_{S,\beta}(u_2)\\
\mathrm{val}_{S,\beta}(\texttt{if } \phi \{u\}) &= \begin{cases} \mathrm{val}_{S,\beta}(u) & \text{for } \mathrm{val}_{S,\beta}(\phi) = tt \\ \emptyset & \text{otherwise} \end{cases}\\
\mathrm{val}_{S,\beta}(\texttt{for } x \{u\}) &= \bigcup \{A(a) \mid a \in U\}
\end{aligned}$$

where $A : U \to (\mathit{Loc} \nrightarrow U)$ is defined by well-founded recursion on $(U, <)$ and the equation $A(a) = \mathrm{val}_{S,\beta^a_x}(u) \oplus \bigcup \{A(b) \mid b \in U,\, b < a\}$

Application of updates: $(S' = S \oplus \mathrm{val}_{S,\beta}(u)$ and $\alpha \in \mathit{Ter} \cup \mathit{For})$

$$\mathrm{val}_{S,\beta}(\{u\}\, \alpha) = \mathrm{val}_{S',\beta}(\alpha)$$

The most involved part of the update evaluation concerns quantified expressions $\texttt{for}\ x\ \{u\}$, whose value is defined by well-founded recursion on $(U, <)$. The definition shows that quantification is a generalisation of parallel composition: informally, for a well-ordered universe $U = \{a < b < c < \cdots\}$ we have

$$\mathrm{val}_{S,\beta}(\texttt{for}\ x\ \{u\}) \;=\; \cdots \oplus \mathrm{val}_{S,\beta^c_x}(u) \oplus \mathrm{val}_{S,\beta^b_x}(u) \oplus \mathrm{val}_{S,\beta^a_x}(u)\ .$$

For a general definition (see Table 1) of the partial interpretation on the right-hand side, we need a union operator on partial functions:[5]

$$\big(\bigcup M\big)(x) = \begin{cases} f(x) & \text{if there is } f \in M \text{ with } f(x) \neq \bot \\ \bot & \text{otherwise} \end{cases}\ ,$$

where we write $f(x) = \bot$ if a partial function f is not defined at point x.

Example 1. The following examples refer to the well-ordered algebra $(\mathbb{N}, <, I)$, where $<$ is the standard order on $\mathbb{N}$. We assume that the vocabulary contains literals and operations $+, \cdot$, and that these symbols are interpreted as usual for $\mathbb{N}$.

$$\mathrm{val}_{S,\beta}(a := 2) \;=\; \{\langle a\rangle \mapsto 2\}$$

In parallel composition, the effect of the left update is invisible to the right one:

$$\mathrm{val}_{S,\beta}(a := 2 \mid f(a) := 3) \;=\; \{\langle a\rangle \mapsto 2,\ \langle f, (\mathrm{val}_{S,\beta}(a))\rangle \mapsto 3\}$$

The right update in parallel composition overrides the left update when clashes occur. Here, this happens for $\mathrm{val}_{S,\beta}(a) = 1$:

$$\mathrm{val}_{S,\beta}(f(a) := 1 \mid f(1) := 2) \;=\; \{\langle f, (1)\rangle \mapsto 2\}$$

In contrast, for $\mathrm{val}_{S,\beta}(a) \neq 1$ both assignments have an effect:

$$\mathrm{val}_{S,\beta}(f(a) := 1 \mid f(1) := 2) \;=\; \{\langle f, (\mathrm{val}_{S,\beta}(a))\rangle \mapsto 1,\ \langle f, (1)\rangle \mapsto 2\}$$

Quantified updates make it possible to define whole functions:

$$\mathrm{val}_{S,\beta}(\{\texttt{for}\ x\ \{f(x) := 2 \cdot x + 1\}\}\ f(5)) \;=\; 11$$

When clashes occur in quantified updates, smaller valuations of the quantified variable will dominate. The smallest individual of $(\mathbb{N}, <)$ is 0:

$$\mathrm{val}_{S,\beta}(\texttt{for}\ x\ \{a := x\}) \;=\; \{\langle a\rangle \mapsto 0\}$$

Update constructors can be nested arbitrarily, like in quantified parallel updates:

$$\mathrm{val}_{S,\beta}(\texttt{for}\ x\ \{(f(x+3) := x \mid f(2 \cdot x) := x + 1)\}) \;=$$
$$\{\langle f, (3)\rangle \mapsto 0, \langle f, (4)\rangle \mapsto 1, \langle f, (5)\rangle \mapsto 2, \cancel{\langle f, (6)\rangle \mapsto 3}, \langle f, (7)\rangle \mapsto 4, \ldots,$$
$$\langle f, (0)\rangle \mapsto 1, \langle f, (2)\rangle \mapsto 2, \cancel{\langle f, (4)\rangle \mapsto 3}, \langle f, (6)\rangle \mapsto 4, \langle f, (8)\rangle \mapsto 5, \ldots\ \}$$

In the last example, both kinds of clashes occur: (i) the pair $\langle f, (6)\rangle \mapsto 3$ stems from $f(x+3) := x$ and is overridden by $\langle f, (6)\rangle \mapsto 4$ (from $f(2 \cdot x) := x + 1$), because updates on the right side of parallel composition dominate updates on the left side ("last-win semantics"). (ii) the pair $\langle f, (4)\rangle \mapsto 3$ stems from the valuation $x \mapsto 2$ and is overridden by $\langle f, (4)\rangle \mapsto 1$ (from $x \mapsto 1$), because small valuations of variables dominate larger valuations ("well-ordered semantics").

[5] The operator $\bigcup$ is obviously not uniquely defined by the given equation, but because of $A(a) \subseteq A(b)$ for $a < b$ its result is unique when defining the evaluation function.

We formalise the behaviour of updates for the latter kind of clashes:

Lemma 1. *Small valuations of variables in updates override larger ones:*

$$\mathrm{val}_{S,\beta}(\texttt{for } x\ \{u\})(loc) \;=\; \mathrm{val}_{S,\beta^m_x}(u)(loc)$$

$$\textit{where}\quad m = \begin{cases} \min_< A & \textit{for } A \neq \emptyset \\ \textit{arbitrary} & \textit{otherwise} \end{cases} \quad \textit{and} \quad A = \{a \mid \mathrm{val}_{S,\beta^a_x}(u)(loc) \neq \bot\}$$

We can now also introduce the equivalence symbol $\equiv$ used in Sect. 2:

Definition 6. *We call two terms, formulas or updates $\alpha_1, \alpha_2 \in Ter \cup For \cup Upd$* equivalent *and write $\alpha_1 \equiv \alpha_2$ if they are necessarily evaluated to the same value: for all well-ordered algebras S and all variable assignments β over S,*

$$\mathrm{val}_{S,\beta}(\alpha_1) = \mathrm{val}_{S,\beta}(\alpha_2)\ .$$

$\equiv$ is a congruence relation for all constructors given in Def. 1 (see Lem. 2).

5 Application of Updates by Rewriting

Updates do in principle not increase the expressiveness of terms or formulas: given an arbitrary term, formula or update α, there will always be an equivalent expression $\alpha' \equiv \alpha$ that does not contain the update application operator.[6] We obtain this result by giving a rewriting system that eliminates updates using altogether 44 rules like $\{u\}\ (t_1 * t_2) \to \{u\}\ t_1 * \{u\}\ t_2$ (with $* \in \{\doteq, \dot{<}\}$). For the complete rewriting system, we have to refer to [3].

Syntactic application of updates to terms or formulas, i.e., simplification of expressions $\{u\}\ \alpha$, is carried out in two phases: first, the update is propagated to subterms or subformulas. In the second phase, when the update has reached a function application, it is analysed whether the update assigns the represented location. For achieving this separation, we need to introduce further operators and extend the syntax given Def. 1 as well as the semantics of Def. 5:

Definition 7. *We define the sets Ter_A, For_A and Upd_A of terms, formulas and updates as in Def. 1, but with further constructors ($x \in Var$ ranges over variables and $f \in \Sigma$ over functions):*

$$\begin{aligned} Ter_A &::= \cdots \mid \{x/Ter_A\}\ Ter_A \mid \textsc{non-rec}(Upd_A, f, (Ter_A, \ldots, Ter_A)) \\ For_A &::= \cdots \mid \{x/Ter_A\}\ For_A \mid \textsc{in-dom}(f, (Ter_A, \ldots, Ter_A), Upd_A) \\ Upd_A &::= \cdots \mid \{x/Ter_A\}\ Upd_A \mid \textsc{reject}(Upd_A, \overline{Upd_A}) \end{aligned}$$

The constructors represent the explicit application of substitutions to terms, formulas, and to updates (like $\{x/s\}\ t$), the non-recursive application of an update u

[6] As we have not formally proven that our rewriting system that turns α into α' is terminating (but consider it as obvious), we do not state this as a theorem.

to function terms $f(\bar{t})$ (like NON-REC$(u, f, \bar{t})$), the test whether an update u assigns to the location denoted by $f(\bar{t})$ (like IN-DOM$(f, \bar{t}, u)$), and filtered updates REJECT$(u_1, \overline{u_2})$ (which are described in Sect. 9). We also extend the evaluation function $\mathrm{val}_{S,\beta}$ on Ter_A, For_A and Upd_A by adding the following clauses:

$$\mathrm{val}_{S,\beta}(\{x/s\}\ \alpha) = \mathrm{val}_{S,\beta'}(\alpha)\ ,$$

where $\beta' = \beta_x^{\mathrm{val}_{S,\beta}(s)}$ and $\alpha \in Ter_A \cup For_A \cup Upd_A$,

$$\mathrm{val}_{S,\beta}(\text{NON-REC}(u, f, \bar{t})) = I'\langle f, \mathrm{val}_{S,\beta}(\bar{t})\rangle\ ,$$

where $S = (U, <, I)$ and $I' = I \oplus \mathrm{val}_{S,\beta}(u)$,

$$\mathrm{val}_{S,\beta}(\text{IN-DOM}(f, \bar{t}, u)) = tt \quad \text{iff} \quad \mathrm{val}_{S,\beta}(u)\langle f, \mathrm{val}_{S,\beta}(\bar{t})\rangle \neq \bot$$

$$\mathrm{val}_{S,\beta}(\text{REJECT}(u_1, \overline{u_2})) = \{(loc \mapsto a) \in \mathrm{val}_{S,\beta}(u_1) \mid \mathrm{val}_{S,\beta}(u_2)(loc) = \bot\}$$

The difference between non-recursive application NON-REC$(u, f, \bar{t})$ and ordinary application $\{u\}\ f(\bar{t})$ is that the subterms $\bar{t}$ are in the first case evaluated in the unmodified algebra, whereas in the latter case the algebra is first updated by u. Formally, we have $\{u\}\ f(\bar{t}) \equiv \text{NON-REC}(u, f, \{u\}\ \bar{t})$. The non-recursive operator enables us to separate the syntactic propagation of updates to subterms and subformulas from the syntactic evaluation of updates.

6 Sequentiality and Application of Updates to Updates

We extend the basic version of updates from Sect. 3 a second time and introduce sequential composition. Sequentiality already occurs when applications of updates are nested, for instance in an expression $\{u_1\}\ \{u_2\}\ \alpha$. It seems natural to make an operator for sequential composition compatible with the nesting of updates: $\{u_1\}\ \{u_2\}\ \alpha \equiv \{u_1\ ;\ u_2\}\ \alpha$. Sequential composition of this kind can be reduced to parallel composition by extending the update application operator to updates themselves, i.e., by considering updates $\{u_1\}\ u_2$.

Definition 8. *We define the sets Ter_{AS}, For_{AS} and Upd_{AS} of terms, formulas and updates as in Def. 7, but with two further constructors:*

$$Upd_{AS} \ ::= \ \cdots \mid Upd_{AS}\ ;\ Upd_{AS} \mid \{Upd_{AS}\}\ Upd_{AS}$$

Again, the evaluation function is extended to Ter_{AS}, For_{AS} and Upd_{AS} by adding two clauses (in both cases $S' = S \oplus \mathrm{val}_{S,\beta}(u_1)$):

$$\mathrm{val}_{S,\beta}(u_1\ ;\ u_2) = \mathrm{val}_{S,\beta}(u_1) \oplus \mathrm{val}_{S',\beta}(u_2), \qquad \mathrm{val}_{S,\beta}(\{u_1\}\ u_2) = \mathrm{val}_{S',\beta}(u_2)$$

The second clause resembles the semantics of update application to terms and formulas. The first clause is very similar to the evaluation of parallel updates, with the only difference that the right update u_2 is evaluated in the structure S' updated by u_1. Intuitively, with parallel composition the effect of u_1 is invisible to u_2 (and vice versa), whereas sequential composition carries out u_1 before u_2. This directly leads to the equivalence $u_1\ ;\ u_2 \equiv u_1 \mid \{u_1\}\ u_2$ that makes it possible to eliminate sequentiality (see [3]).

The relation $\equiv$ from Def. 6 can be extended to Ter_{AS}, For_{AS} and Upd_{AS}:

Lemma 2. *Equivalence $\equiv$ of terms, formulas and updates is a congruence relation for all constructors given in Def. 1, 7 and 8.*

Example 2. We continue Example 1 and assume the same vocabulary/algebra.

$$\begin{aligned} a := 1\,;\ f(a) := 2 \quad &\equiv \quad a := 1 \mid f(1) := 2 \\ \mathrm{val}_{S,\beta}(a := 1\,;\ f(a) := 2) \quad &= \quad \{\langle a\rangle \mapsto 1,\ \langle f,(1)\rangle \mapsto 2\} \\ \mathrm{val}_{S,\beta}(a := 1\,;\ (a := 3 \mid f(a) := 2)) \quad &= \quad \{\langle a\rangle \mapsto 3,\ \langle f,(1)\rangle \mapsto 2\} \end{aligned}$$

7 Modelling Heap Structures

The memory of imperative and object-oriented programs can be modelled as a well-ordered algebra by choosing appropriate vocabularies Σ. By updating the values of function symbols, the memory contents can be modified symbolically. Compared to a more explicit encoding of program states as individuals (for instance, elements of a datatype), directly representing memory using a first-order vocabulary leads to very readable formulas that are in particular suited for interactive proof systems (see [3] for a more detailed discussion).

In the whole section, we assume that the universe for evaluating updates are the natural numbers $\mathbb{N}$, and that the standard well-ordering $<$ is used (as in Example 1). A more realistic application would, of course, require a typed logic and to model the datatypes of programming languages properly. For this section, it shall suffice to treat both data and addresses/pointers as natural numbers.

Variables: The simplest way to store data in programs is the usage of global variables, which can be seen as constants $g, h, i, \ldots \in \Sigma$ when representing program memory using well-ordered algebras ($\alpha(g) = \alpha(h) = \cdots = 0$). Assignments are naturally performed through updates $g := t$. Expanding a sequential update into a parallel update yields a representation of the post-state by describing the post-values of all modified variables in terms of the pre-values:[7]

$$\mathit{gswap} \quad = \quad i := g\,;\ g := h\,;\ h := i \quad \equiv \quad g := h \mid h := g \mid i := g$$

Classes and Attributes: The individual objects of a class can be distinguished using addresses (natural numbers). Instance attributes of a class C are then unary functions $a_C, b_C \ldots \in \Sigma$ (with $\alpha(a_C) = \alpha(b_C) = \cdots = 1$) that take an address as argument. As an example, we consider again the class *List* representing doubly-linked lists from Sect. 2 (with attributes $\mathit{next}, \mathit{prev}, \mathit{val} \in \Sigma$). The following two updates describe the setup of singleton lists (that hold a value v) and the concatenation of two lists (where one list ends with the object e and the second one begins with the object b):

$$\begin{aligned} \mathit{setup}(o, v) \quad &= \quad o.\mathit{prev} := \mathit{nil} \mid o.\mathit{val} := v \mid o.\mathit{next} := \mathit{nil} \\ \mathit{cat}(e, b) \quad &= \quad e.\mathit{next} := b \mid b.\mathit{prev} := e \end{aligned}$$

[7] We leave out parentheses because both parallel and sequential composition are associative, see (R52) and (R53) in Table 2.

(we assume that $nil \in \Sigma$ denotes invalid addresses and the beginning and end of lists). The update *init* from Sect. 2 and a list containing the numbers $0, \ldots, n$ can then be set up as follows:

$$\begin{aligned} \mathit{init} &\equiv \mathit{setup}(a,c)\,;\ \mathit{setup}(b,2)\,;\ \mathit{cat}(a,b)\,;\ a.\mathit{next}.\mathit{val} := d \\ \mathit{seq} &= \texttt{for}\ x\ \{\texttt{if}\ x \doteq n+1\ \{\mathit{setup}(x,x)\}\}\,;\ \texttt{for}\ x\ \{\texttt{if}\ x \doteq n\ \{\mathit{cat}(x,x+1)\}\} \\ &\equiv_{\mathbb{N}} 0.\mathit{prev} := \mathit{nil} \mid n.\mathit{next} := \mathit{nil} \mid \texttt{for}\ x\ \{\texttt{if}\ x \doteq n+1\ \{x.\mathit{val} := x\}\} \mid \\ &\quad \texttt{for}\ x\ \{\texttt{if}\ x \doteq n\ \{x.\mathit{next} := x+1\}\} \mid \\ &\quad \texttt{for}\ x\ \{\texttt{if}\ x \doteq n\ \{(x+1).\mathit{prev} := x\}\} \end{aligned}$$

Properties about the lists can be proven by applying the updates and performing first-order reasoning:

$$\forall x.\,(\neg x \doteq n \vee \{\mathit{seq}\}\ x.\mathit{next}.\mathit{prev} \doteq x) \quad \equiv_{\mathbb{N}} \quad \forall x.\,(\neg x \doteq n \vee x \doteq x) \quad \equiv \quad \mathit{true}$$

Object Allocation: Updates cannot add or remove individuals from a universe (*constant-domain semantics*). In modal logic, the usual way to simulate changing universes is to introduce a predicate that distinguishes between existing and non-existing individuals. Likewise, for our heap model "implicit" attributes $\mathit{created}_C$ can be defined that, for instance, have value 1 for existing and 0 for non-existing objects of a class C. An initial state in which no objects are allocated can be reached through the update $\texttt{for}\ x\ \{x.\mathit{created}_C := 0\}$. We write an allocator for list nodes as follows:[8]

$$\mathit{alloc}(o,v) \quad = \quad o := \min i.\,(i.\mathit{created}_{\mathit{List}} \doteq 0)\,;\ \big(o.\mathit{created}_{\mathit{List}} := 1 \mid \mathit{setup}(o,v)\big)$$

Note, that allocating objects in parallel using this method will produce clashes, because parallel updates cannot observe each other's effects. When running in parallel, $\mathit{alloc}(a,1)$ and $\mathit{alloc}(b,2)$ will deterministically allocate the same object:

$$\mathit{alloc}(a,1) \mid \mathit{alloc}(b,2) \quad \equiv \quad \mathit{alloc}(b,2)\,;\ a := b \quad \not\equiv \quad \mathit{alloc}(a,1)\,;\ \mathit{alloc}(b,2)$$

Arrays: Arrays in a Java-like language behave much like objects of classes, with the difference that arrays provide numbered cells instead of attributes. We can model arrays be introducing a binary access function $ar \in \Sigma$ and a unary function $\mathit{len} \in \Sigma$ telling the length of arrays ($\alpha(ar) = 2$ and $\alpha(\mathit{len}) = 1$). Array allocation can be treated just like allocation of objects through an implicit attribute $\mathit{created}_{ar}$. Given this vocabulary, we can allocate an array of length n and fill it with numbers $0, \ldots, n-1$: (we write $o[x]$ instead of $ar(o,x)$)

$$\begin{aligned} \mathit{alloc}_{ar}(o,n) &= o := \min i.\,(i.\mathit{created}_{ar} \doteq 0)\,;\ \big(o.\mathit{created}_{ar} := 1 \mid o.\mathit{len} := n\big) \\ \mathit{seq}_{ar} &= \mathit{alloc}_{ar}(o,n)\,;\ \texttt{for}\ x\ \{\texttt{if}\ x \doteq o.\mathit{len}\ \{o[x] := x\}\} \end{aligned}$$

[8] For practical purposes, it is reasonable to have more book-keeping about allocated objects than shown here. The approach that is followed in KeY is to introduce variables $\mathit{nextToCreate}_C$ and to allocate objects sequentially.

8 Symbolic Execution in Dynamic Logic Revisited

As shown in Sect. 2, during symbolic execution, updates can represent a certain prefix (or path) of a program, whereas the suffix that remains to be executed is given in the original language. In order to use updates for symbolic execution, first of all a suitable representation of the program states using a first-order vocabulary and algebras (along the lines of Sect. 7) has to be chosen. Rewriting rules then define the semantics of program features in terms of updates and of connectives of first-order logic. This approach has been used to implement symbolic execution for the "real-world" language JavaCard [8]. Examples for the rewriting rules are:[9]

$$\langle\,\rangle\,\phi \quad\rightsquigarrow\quad \phi, \qquad\qquad \langle\, s = t;\ \alpha\,\rangle\,\phi \quad\rightsquigarrow\quad \{s := t\}\,\langle\,\alpha\,\rangle\,\phi$$
$$\langle\,\texttt{if}\ (b)\ \beta_1;\ \texttt{else}\ \beta_2;\ \alpha\,\rangle\,\phi \quad\rightsquigarrow\quad (b \wedge \langle\,\beta_1;\ \alpha\,\rangle\,\phi) \vee (\neg b \wedge \langle\,\beta_2;\ \alpha\,\rangle\,\phi)$$

It is important to note that updates are *not* intended as an intermediate representation for complete programs: the focus is on handling the sequential parts. For reasoning about general loops or recursion, techniques like induction or invariants are still necessary. It is, nevertheless, possible to translate certain loops directly to an update [9]. An example are many array operations in Java:[10]

$$\langle\,\texttt{System.arrayCopy}(ar_1, o_1, ar_2, o_2, n)\,\rangle\,\phi$$
$$\rightsquigarrow\quad \{\texttt{for}\ x\ \{\texttt{if}\ \neg x \doteq o_2 \wedge x \doteq o_2 + n\ \{ar_2[x] := ar_1[x - o_2 + o_1]\}\}\}\,\phi$$

Compared to a declarative specification of `arrayCopy` using a post-condition that contains a universally quantified formula, the imperative update can be applied to formulas or terms like a substitution. We consider updates as advantageous both for interactive and automated reasoning: the program structure is preserved, and unnecessary non-determinism in a derivation is avoided.

A characteristic of imperative programs is that memory locations can be assigned to/overwritten multiple times. After elimination of sequential composition, overwritten locations occur as clashes in updates. An example is the update *init* from Sect. 2 and 7, which contains potential clashes because of aliasing: for $a \doteq b$, the expressions $a.val$ and $b.val$ denote the same location. Due to last-win semantics, it is not necessary to distinguish the possible cases when turning sequential composition into parallel composition. Only when applying the update, as in the expression co' in Sect. 2, the case $a \doteq b$ has to be handled explicitly.

Well-ordered semantics enables an implicit handling of output dependencies in loops (different iterations assign to the same locations) in a similar way [9]. A simple example is: ($e(i)$ is a side-effect free, possibly non-injective expression)

$$\langle\,\texttt{while}\ \ (\neg i \doteq 0)\ \ \{i = i - 1;\ a[e(i)] = i;\}\,\rangle\,\phi$$
$$\rightsquigarrow\quad \{i := 0 \mid \texttt{for}\ x\ \{\texttt{if}\ x \doteq i\ \{a[e(x)] := x\}\}\}\,\phi$$

[9] s, t, b have to be free of side-effects. It general, it will also be necessary to define a translation of side-effect free program expressions into terms.

[10] For sake of clarity, the example ignores the diverse errors that can occur when calling `arrayCopy`, for instance for $ar_1 \doteq ar_2$.

Table 2. Laws for Commuting and Distributing Update Connectives

For $\alpha \in \mathit{Ter}_{AS} \cup \mathit{For}_{AS} \cup \mathit{Upd}_{AS}$:

$$\{u_1\}\,\{u_2\}\,\alpha \equiv \{u_1\,;\,u_2\}\,\alpha \tag{R51}$$

$$u_1 \mid (u_2 \mid u_3) \equiv (u_1 \mid u_2) \mid u_3 \tag{R52}$$

$$u_1\,;\,(u_2\,;\,u_3) \equiv (u_1\,;\,u_2)\,;\,u_3 \tag{R53}$$

$$u_1 \mid u_2 \equiv \textsc{reject}(u_1, \overline{u_2}) \mid u_2 \tag{R54}$$

$$u_1 \mid u_2 \equiv u_2 \mid \textsc{reject}(u_1, \overline{u_2}) \tag{R55}$$

$$u \equiv u \mid \texttt{if}\ \phi\ \{u\} \qquad (\phi \text{ arbitrary}) \tag{R56}$$

$$u_1 \equiv u_1 \mid \textsc{reject}(u_1, \overline{u_2}) \qquad (u_2 \text{ arbitrary}) \tag{R57}$$

$$\texttt{if}\ \phi\ \{u_1 \mid u_2\} \equiv \texttt{if}\ \phi\ \{u_1\} \mid \texttt{if}\ \phi\ \{u_2\} \tag{R58}$$

$$\texttt{if}\ \phi_1\ \{\texttt{if}\ \phi_2\ \{u\}\} \equiv \texttt{if}\ \phi_1 \wedge \phi_2\ \{u\} \tag{R59}$$

$$\texttt{for}\ x\ \{\texttt{if}\ \phi\ \{u\}\} \equiv \texttt{if}\ \phi\ \{\texttt{for}\ x\ \{u\}\} \qquad (x \notin \mathrm{fv}(\phi)) \tag{R60}$$

$$\texttt{for}\ x\ \{\texttt{if}\ \phi\ \{u\}\} \equiv \texttt{if}\ \exists x.\,\phi\ \{u\} \qquad (x \notin \mathrm{fv}(u)) \tag{R61}$$

$$\texttt{for}\ x\ \{u_1 \mid u_2\} \equiv \texttt{for}\ x\ \{u_1\} \mid u_2 \qquad (x \notin \mathrm{fv}(u_2)) \tag{R62}$$

For $u = \texttt{for}\ z\ \{\texttt{if}\ z \dot{<} x\ \{\{x/z\}\ u_1\}\}$ and $z \neq x$, $z \notin \mathrm{fv}(u_1)$:

$$\texttt{for}\ x\ \{u_1\} \equiv \texttt{for}\ x\ \{\textsc{reject}(u_1, \overline{u})\} \tag{R63}$$

$$\texttt{for}\ x\ \{u_1 \mid u_2\} \equiv \texttt{for}\ x\ \{u_1\} \mid \texttt{for}\ x\ \{\textsc{reject}(u_2, \overline{u})\} \tag{R64}$$

For $u = \texttt{for}\ z\ \{\texttt{if}\ z \dot{<} x\ \{\{x/z\}\ \texttt{for}\ y\ \{u_1\}\}\}$ and $|\{x, y, z\}| = 3$, $z \notin \mathrm{fv}(u_1)$:

$$\texttt{for}\ x\ \{\texttt{for}\ y\ \{u_1\}\} \equiv \texttt{for}\ y\ \{\texttt{for}\ x\ \{\textsc{reject}(u_1, \overline{u})\}\} \tag{R65}$$

9 Laws for Update Simplification

Sect. 7 demonstrates how updates can be simplified and written as parallel composition of assignments. More formally, we can extend Sect. 5 and state that, given an arbitrary update u, there will always be an equivalent update $u' \equiv u$ of the following shape: (in which ϕ_i, s_i, t_i do not contain further updates)

$$\begin{array}{l} \texttt{for}\ x_{1,1}\ \{\texttt{for}\ x_{1,2}\ \{\texttt{for}\ \cdots\ \{\texttt{if}\ \phi_1\ \{s_1 := t_1\}\}\}\} \\ \mid \cdots \\ \mid \texttt{for}\ x_{k,1}\ \{\texttt{for}\ x_{k,2}\ \{\texttt{for}\ \cdots\ \{\texttt{if}\ \phi_k\ \{s_k := t_k\}\}\}\} \end{array} \tag{1}$$

It is usually advantageous to establish this shape: (i) Obvious clashes, like in the update $g := 1 \mid g := 2$, can easily be eliminated. (ii) The update can easily be read and directly tells about the values of variables or heap contents. (iii) When applying updates syntactically using the rewriting system of Sect. 5, this form is more efficient than most other shapes, because it supports the search for matching assignments. (iv) It is possible to define more specialised and efficient rewriting rules for update application (than the ones given in [3]). This has been done for the implementation of updates in KeY.

Table 2 gives, besides others, identities that enable to establish form (1) by turning sequential composition into parallel composition, distributing `if` and `for` through parallel composition and commuting `if` and `for`. In this table, we denote the set of free variables of an expression α with $\mathrm{fv}(\alpha)$ (see, e.g., [5]). The soundness of all rules and identities, based on the semantics of Sect. 4, has been proven using the Isabelle/HOL proof assistant.

For formulating the transformation rules, we need a further operator from Def. 7: the expression $\textsc{reject}(u_1, \overline{u_2})$ denotes an update that carries out exactly those assignments of u_1 that do *not* define locations that are also assigned to by u_2. This enables us to make updates disjoint, i.e., to prevent updates from assigning to the same locations, which is often a premise for permuting updates. Disjointness is relevant for parallel composition (R55) and for quantification (R64), (R65), where permutation can change the order of assignments.

10 Related Work

A theory that is very similar to updates are abstract state machines (ASMs) [10]. While there are different versions of ASMs, all update constructors of this paper can in similar form also be found in [11]. The main difference is the notion of "consistent updates" that exists for ASMs and that demands clash-freeness. In contrast, the present paper describes a semantics in which clashes are resolved by a last-win strategy or a well-ordering strategy, which we consider as better suited for representing imperative programs.

Substitutions in B [12] have character similar to updates. Like ASMs, they are used for modelling systems and are a complete programming language that also provides loops and non-determinism. Updates are deliberately kept less expressive, focussing on automated simplification and application.

The guarded command language [13] is used as intermediate language in the verification systems ESC/Java2 and Boogie. In contrast to updates, guarded commands are used to represent *complete* object-oriented programs—which requires concepts like loops or non-determinism—and are eliminated using wp-calculus.

In the context of the KeY system, updates turn up in [8] for the first time, where the only update constructor are assignments. Parallel updates are described in [14,15] for the first time, and have the same last-win semantics as in this paper.

11 Conclusions and Future Work

The update language described in this paper has been implemented in the KeY prover. Quantified updates, added most recently, have mostly improved the ability of the prover to handle arrays, as operations like `arrayCopy` (Sect. 8) can now be specified and symbolically executed very efficiently. Compared to the rules in Sect. 5 and 9 (which are more general), KeY also contains further optimisations for applying updates that have been found to be important in practice.

In the future, an interesting step would be the combination of ordinary substitutions and updates. This would require developing a concept of bound renaming for updates. Another appealing improvement would be the possibility of non-deterministic updates, which would allow to handle object creation (or, generally, under-specification of language features) more naturally.

Acknowledgements

I want to thank Reiner Hähnle for bringing up the idea of extending the update language by adding quantification, as well as for discussions. I am also grateful for discussions and comments from Wolfgang Ahrendt, Richard Bubel, and Steffen Schlager, and for comments from the anonymous referees.

References

1. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press (2000)
2. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY Tool. Software and System Modeling **4** (2005) 32–54
3. Rümmer, P.: Proving and disproving in dynamic logic for Java. Licentiate Thesis 2006–26L, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden (2006)
4. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
5. Fitting, M.C.: First-Order Logic and Automated Theorem Proving. 2nd edn. Springer-Verlag, New York (1996)
6. Zermelo, E.: Beweis dass jede Menge wohlgeordnet werden kann. Mathematische Annalen **59** (1904) 514–516
7. Spivey, J.M.: The Z Notation: A Reference Manual. 2nd edn. Prentice Hall (1992)
8. Beckert, B.: A dynamic logic for the formal verification of JavaCard programs. In Attali, I., Jensen, T., eds.: Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France. Volume 2041 of LNCS., Springer (2001) 6–24
9. Gedell, T., Hähnle, R.: Automating verification of loops by parallelization. In: Proceedings, 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning. LNAI, Springer (2006) To appear.
10. Gurevich, Y.: Evolving Algebras 1993: Lipari Guide. In Börger, E., ed.: Specification and Validation Methods. Oxford University Press (1995) 9–36
11. Stärk, R.F., Nanchen, S.: A logic for abstract state machines. Journal of Universal Computer Science **7** (2001) 981–1006
12. Abrial, J.R.: The B Book: Assigning Programs to Meanings. Cambridge University Press (1996)
13. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)
14. Platzer, A.: An object-oriented dynamic logic with updates. Master's thesis, University of Karlsruhe, Department of Computer Science. Institute for Logic, Complexity and Deduction Systems (2004)
15. Beckert, B., Platzer, A.: Dynamic logic with non-rigid functions: A basis for object-oriented program verification. In Furbach, U., Shankar, N., eds.: Proceedings, IJCAR, Seattle, USA. LNCS, Springer (2006) To appear.