Specification Predicates with Explicit Dependency Information

Richard Bubel[1], Reiner Hähnle[1], and Peter H. Schmitt[2]

[1] Dept. of Computer Science and Engg., Chalmers Univ. of Technology
bubel|reiner@chalmers.se
[2] Dept. of Computer Science, Univ. of Karlsruhe
pschmitt@ira.uka.de

Abstract. Specifications of programs use auxiliary symbols to encapsulate concepts for a variety of reasons: readability, reusability, structuring and, in particular, for writing recursive definitions. The definition of these symbols often depends implicitly on the value of other locations such as fields that are not stated explicitly as arguments. These hidden dependencies make the verification process substantially more difficult. In this paper we develop a framework that makes dependency on locations explicit. This allows to define general simplification rules that avoid unfolding of predicate definitions in many cases. A number of non-trivial case studies show the usefulness of the concept.

1 Introduction

In program logics, especially in logics that target object-oriented languages, state-dependent predicates or functions are a convenient and often necessary concept used in specifications. They allow one to keep specifications concise and easy to read for humans. They are indispensable for the specification of inherently recursive properties such as reachability. Especially in first-order program logics there is no other alternative to specify properties recursively.

Such state-dependent predicate or function symbols, which are sometimes called *non-rigid symbols*, are not straightforward to use in verification practice, because they require special inference techniques. To unpack and transform their definition after every single state change would be extremely inefficient and must be avoided. As a first example, we consider the frequent specification task that stipulates an object array a to contain only non-null references. It is convenient to define a non-rigid unary predicate symbol on arrays:

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

A typical desirable property in this context is that a simple assignment to a program variable `j` does not change the validity of nonNullArray. The formulation in Hoare logic [11] is in the first line below, the second line is the same in Dijkstra's weakest precondition calculus [9], and the third line reformulates it in Dynamic Logic [10]:

$$\begin{array}{l}\{\mathrm{nonNullArray}(a)\}\ \mathtt{j} := \mathtt{j} + \mathtt{1};\ \{\mathrm{nonNullArray}(a)\} \\ \mathrm{nonNullArray}(a)\ \rightarrow\ wp(\mathtt{j} := \mathtt{j} + \mathtt{1},\ \mathrm{nonNullArray}(a)) \\ \mathrm{nonNullArray}(a)\ \rightarrow\ \langle \mathtt{j} := \mathtt{j} + \mathtt{1}; \rangle \mathrm{nonNullArray}(a)\ \ .\end{array}$$

To prove this claim naïvely entails unpacking the definition of nonNullArray or to define a special-purpose predicate transformer having specific knowledge about the state dependencies in its definition. For example, the definition of nonNullArray contains an implicit dependency on a and on $a[i]$ for all i, but not on any integer program variable `j`. The contribution of this paper is a technique that makes this kind of implicit dependency information explicit in the symbol's syntax. This leads to significantly higher automation. The presented approach has been evaluated in several small examples, e.g. verification of a selection sort algorithm, but also to verify an implementation of the Schorr-Waite algorithm. It allows to formulate *uniform* predicate transformers that can exploit generic dependencies and that are applicable in a variety of situations. This paper is partly based on results first published in the first co-author's dissertation [8].

The paper is structured as follows. The program logic used in this paper is introduced in Sect. 2. Syntax and semantics for the explicit dependency notation called *location descriptors* are presented in Sect. 3. In Sect. 4 we show how to prove that the chosen location descriptor is consistent with the predicate's definition. Predicate transformers in the form of simplification rules that take advantage of explicit dependencies are presented in Sect. 5. Sect. 6 presents applications and case studies that show the usefulness of the concept of location descriptors. Sect. 7 credits related work and outlines future work.

2 Dynamic Logic with Updates

This section sketches the program logic used throughout the paper. We use object-oriented dynamic logic (ODL) [4] which extends standard dynamic logic [10] to cover all essential features of object-orientation, but is small enough for theoretical purposes.

The programming language used in ODL is essentially a stripped down version of Java. It supports all concepts except for inner and anonymous classes, floating point arithmetic, threads (and, hence, GUI). Additionally, ODL does not support dynamic method binding or exceptions. Some of these restrictions stem from open scientific problems (floating points, threads), others are a consequence of the goal to define a minimalistic object-oriented language into which all aspects of realistic languages can be compiled without much overhead. We concentrate on the language aspects essential for the paper and leave out, for example, object allocation or exceptions. For a full account see [4]. When convenient we use the style of presentation given in [3] for the logic Java Card DL.

Definition 1 (Signature). *Let $\mathcal{T}_{all} := \{\top, boolean, int, \bot\} \cup \mathcal{T}_d$ denote a finite set of types. $(\mathcal{T}_{all}, \sqsubseteq)$ forms a complete lattice with respect to the partial order $\sqsubseteq$ (modelling the subtype hierarchy). $\mathcal{T}_d$ is the set of all reference types (closed by intersection) containing the* `Null` *type as least element. Further, $\mathcal{T} := \mathcal{T}_{all} - \{\bot\}$ denotes the set of all types except the bottom type.*

$\Sigma_{\mathcal{T}} := \{Q, Op, Mod, U, \Pi, \mathrm{PSym}, \mathrm{FSym}, \mathrm{VSym}\}$ *is a typed signature over* $\mathcal{T}$*, where* Q, Op *are sets containing the classical first-order quantifiers and operators.* Mod *contains the box* $[\cdot]$ *and diamond* $\langle\cdot\rangle$ *modalities.* U *is the set of elementary updates (defined below) and* Π *the set of programs.* PSym, FSym *are typed predicate and function symbols with arity function* α *assigning each symbol its signature.* VSym *is a set of typed first-order logic variables.*

As mentioned in Sect. 1 we distinguish between rigid and state-dependent (non-rigid) function and predicate symbols with the following notation: $\mathrm{PSym} = \mathrm{PSym}_r \cup \mathrm{PSym}_{nr}$ and $\mathrm{FSym} = \mathrm{FSym}_r \cup \mathrm{FSym}_{nr}$. It is useful to single out from the non-rigid function symbols FSym_{nr} the subset of location function symbols, i.e., functions used to represent local program variables, fields and arrays that can be changed by a program or update:

Definition 2 (Location Function Symbols). *The set of* location function symbols $\mathrm{FSym}_{loc} \subseteq \mathrm{FSym}_{nr}$ *contains for each arity an infinite number of symbols including*

- *for each program variable pv used in an ODL program a constant symbol with the same name and type;*
- *for each attribute* a *of type* T *declared in a class* C *of the ODL program, a unary location function symbol* $a@(C) : C \rightarrow T$*;*
- *the array access operator* $[] : \top \times int \rightarrow \top$.

Note 1. ODL features separate syntactic categories for program and logic (first-order) variables. Program variables are modelled as non-rigid constants and cannot be quantified. Logic variables are rigid, quantifiable, and must not occur inside programs. We omit the $@(C)$ suffix from attribute names if no ambiguity arises and write $o.a$ instead of $a(o)$ for attribute lookup expressions.

Definition 3 (Updates). *Given a location function* f *and terms* o_i *and* v*. Then the expression* $f(o_1, \ldots, o_n) := v$ *denotes an* elementary update*. General updates are defined inductively. Let* U*,* U_1, U_2 *be updates, then all of the following expressions are updates as well:*

- *the sequential composition* $U_1 \,;U_2$*,*
- *the parallel composition* $U_1 \,||\, U_2$*,*
- *the conditional* `\if` (ϕ); U *(where* ϕ *is a formula), and*
- *the quantification* `\for` x; U *(binding the first-order variable* x *in update* U*).*

Definition 4 (Terms and Formulae). *The inductive definition of terms and formulae is as usual for typed first-order dynamic logic. We define only the less common cases:*

- *Let* U *be an update and* ξ *a term (formula), then* $\{U\}\ \xi$ *is a term (formula).*

– *Let ϕ be a formula and p a program, then $[p]\phi$ (partial correctness) and $\langle p\rangle\phi$ (total correctness) are formulae.*

The following definitions formalise the semantics of formulae and updates.

Definition 5 (ODL Kripke Structure). *An ODL Kripke structure $\mathcal{K} := (\mathcal{M}, \mathcal{S}, \rho)$ consists of*

– *A partial first-order model $\mathcal{M} = (\mathcal{D}, I_0)$ providing a domain mapping $\mathcal{D}$ that assigns to each type its domain (with $\mathcal{D}(int) = \mathbb{Z}$) and a partial interpretation I_0 for all rigid and non-rigid predicate symbols:*

$$I_0(q) := \begin{cases} \uparrow \ , & \text{if } q \in \mathrm{PSym}_{nr} \\ G \ , & \text{if } q \in \mathrm{PSym}_r, \text{ for some } G \subseteq \mathcal{D}(T_1) \times \ldots \times \mathcal{D}(T_n) \end{cases}$$

where $q : T_1 \times \ldots \times T_n$ is a predicate symbol (analogous for function symbols)

– *A set of states $\mathcal{S}$ where each $S \in \mathcal{S}$ contains an interpretation I_{nr} completing the partial interpretation I_0 to a total interpretation $I := I_0 \dot\cup I_{nr}$ by assigning a meaning to all non-rigid symbols.*

– *The state transition relation ρ defining the programs' semantics, where for a program p and two states $S_1, S_2 \in \mathcal{S}$ the relation $\rho(p)(S_1, S_2)$ holds if and only if executing p in state S_1 terminates in the final state S_2. As ODL programs are deterministic the final state is unique whenever it exists.*

A function $\beta : \mathrm{VSym} \to \bigcup_{T \in \mathcal{T}} \mathcal{D}(T)$ assigning to all logic variables an element of the universe of the appropriate type is a variable assignment*.*

Definition 6 (Semantic Location). *A* semantic location *is defined as a tuple $\langle f, (e_1, \ldots, e_n)\rangle$ where $f : T_1 \times \ldots \times T_n \to T$ is a location function symbol as in Def. 2 and e_i (for $i \in \{1, \ldots, n\}$) are elements in $\mathcal{D}(T_i)$.*

Definition 7 ((Consistent) Semantic Update). *An* elementary semantic update *is a pair $(\langle f, (e_1, \ldots, e_n)\rangle, d)$ where $\langle f, (e_1, \ldots, e_n)\rangle$ is a semantic location with $f : T_1 \times \ldots \times T_n \to T$ and d an element of $\mathcal{D}(T)$. A (possible empty) set of elementary semantic updates is called* semantic update*. A semantic update is called* consistent*, if it contains for any semantic location at most one* elementary semantic update*.*

Definition 8 (Application of a Consistent Semantic Update). *The application of a consistent semantic update CU is a mapping between states. Applying CU in a state S maps it to the state $S' = CU(S)$ that coincides on the value of all location function with S except for the semantic locations occurring in CU: whenever $(\langle f, (e_1, \ldots, e_n)\rangle, d) \in CU$ then $S'(f)(e_1, \ldots, e_n)$ evaluates to d.*

Updates are an explicit notation to capture symbolic state changes. The definition of a semantics for updates is rather technical due to clashes when an update assigns different values to the same location. Sequential composition describes two successive state changes, while parallel composition updates the locations simultaneously and may cause clashes by updating the same location with differing values, e.g., $l := t_1 \,||\, l := t_2$. We use a last-one-wins clash resolution, i.e., in the resulting state location l has the value t_2. Quantified updates allow us to represent state changes of infinitely many locations. Resolution of clashes caused by quantified updates is left out for space reasons (see [3, 18]).

Example 1. Some updates and their intended semantics:

– The elementary update $\mathtt{x} := t$ assigns to the program variable $\mathtt{x}$ the value t. Applying it to a program variable $\mathtt{y}$ in $\{\mathtt{x} := t\}\ \mathtt{y}$ results in t if $\mathtt{x}$ and $\mathtt{y}$ are the same variable, otherwise, the term evaluates to $\mathtt{y}$.
– The parallel update $\mathtt{x} := \mathtt{y} \,||\, \mathtt{y} := \mathtt{x}$ swaps the content of the program variables $\mathtt{x}$ and $\mathtt{y}$. Parallel updates are evaluated simultaneously and, therefore, are independent of each other. More formally, let $\mathcal{K} := (\mathcal{M}, \mathcal{S}, \rho)$ be an ODL Kripke structure, $S \in \mathcal{S}$ and β a variable assignment. Evaluating the above update under $(\mathcal{K}, S, \beta)$ results in the consistent update CU mapping S to a state $CU(S)$ coinciding with S except for the program variables $\mathtt{x}$ and $\mathtt{y}$ whose values in $(\mathcal{K}, S, \beta)$ are swapped.
– The quantified update $\backslash\mathtt{for}\ i;\ a[i] := 0$ assigns 0 to all components of a.

We continue by defining the semantics of terms and formulae:

Definition 9 (Semantics of Terms and Formulae). *The inductive semantic definitions are as usual. We list only a few non-obvious cases, full definitions are in [3, 4].*

Let $\mathcal{K} := (\mathcal{M}, \mathcal{S}, \rho)$ denote an ODL Kripke structure, $t^{(\mathcal{K},S,\beta)}$ the evaluation of term t in state S under a variable assignment β, and $\models$ the validity relation.

– $(\{\mathcal{U}\}\ t)^{(\mathcal{K},S,\beta)} := t^{(\mathcal{K},S',\beta)}$ *where* $S' = \mathcal{U}^{(\mathcal{K},S,\beta)}(S)$
– $S, \beta \models \langle p \rangle \phi\ (p \in \Pi)$ *iff a state* S' *exists with* $S', \beta \models \phi$ *and* $\rho(p)(S, S')$
– $S, \beta \models [p]\phi\ (p \in \Pi)$ *iff* $S', \beta \models \phi$ *holds for any state* S' *with* $\rho(p)(S, S')$
– $S, \beta \models \{\mathcal{U}\}\ \phi$ *iff* $S', \beta \models \phi$*, where* $S' = \mathcal{U}^{(\mathcal{K},S,\beta)}(S)$

Later in the paper we need means to relate two arbitrary states on the syntactic level. For this we use a special kind of update called *anonymous program update* whose purpose is to perform a state transition to an unspecified state. Anonymous programs of this kind are well-known from propositional dynamic logic [10]. They are deterministic and terminating.

Definition 10 (Anonymous Program, Anonymous Program Update). *The atomic programs st_1, $st_2 \ldots$ are called (elementary)* anonymous programs*. The set of programs Π is extended accordingly. Further, we extend the inductive definition of updates by including the elementary* anonymous program update *ω_i for each anonymous program st_i.*

Definition 11 (Semantics of Anonymous Program Updates). *An anonymous program st_i is interpreted in an ODL Kripke Structure $\mathcal{K}$ such that for all $S \in \mathcal{S}$ there exists exactly one state $S' \in \mathcal{S}$ such that $\rho(st_i)(S, S')$ holds. An* anonymous program update *ω_i is then evaluated to a state transformer such that $\omega_i^{(\mathcal{K},S,\beta)}(S) = S'$ for all variable assignments β.*

3 Symbols with Explicit Dependencies

In this section we introduce a syntactic notation for non-rigid symbols that renders explicit the implicit dependencies on the state in their definition.

3.1 Location Descriptors

We need a notation to describe sets of locations. We use *location descriptors* introduced in the KeY-system [3] for modifies/assignable clauses. The origin of this notation goes back to quantified updates [18], see also Sect. 2. Location descriptors permit a compact and extensional characterisation of location sets.

Definition 12 (Location Descriptor). *A* location descriptor *has the form*

$$\texttt{\textbackslash for}\ x_1, \ldots, x_n; \texttt{\textbackslash if}\ (\Phi)\ loc$$

where (i) $x_1, \ldots, x_n$ are variables bound in Φ and loc, (ii) Φ is an arbitrary formula, and (iii) loc is a term with a location function symbol as top level operator, i.e., a program variable, an attribute function or the array access function. Except for $x_1, \ldots, x_n$ no other free *variables occur in Φ or loc.*

Location descriptors ld_1, ld_2 can be accumulated to sets of location descriptors by concatenation: $ld_{new} := ld_1\ ;\ ld_2$. In case no variables are bound or Φ is identical to true, the corresponding parts in the syntax can be omitted.

Example 2. Here are some typical usages of location descriptors:

– `\for List` x; x`.next` capturing all `next` locations of `List`-typed elements. Note that the guard has been omitted here.
– `\for` $T[]\ a$, int i;`\if` $(i >= 0 \land i < a.\texttt{length})$ $a[i]$ meaning all T-typed array component locations with indexes between 0 and the array length.[3]
– `\for Tree` t; t`.left` ;`\for Tree` t; t`.right` capturing all `left` and `right` locations of `Tree`-typed elements.

Definition 13 (Location Descriptor Extension). *Let $\mathcal{K} := (\mathcal{M}, \mathcal{S}, \rho)$ be an ODL Kripke structure with universe $\mathcal{D} = \bigcup_{T \in \mathcal{T}} \mathcal{D}(T)$. The* extension of a location descriptor *$ld =$ `\for` $x_1, \ldots, x_n$;`\if` $(\Phi)\ l(t_1, \ldots, t_n)$ in a given state $S \in \mathcal{S}$ is defined as the set of semantic locations:*

$$ld^{(\mathcal{K},S)} := \{\langle\, l,\ (t_1, \ldots, t_n)^{(\mathcal{K},S,\beta)} \rangle \mid (\mathcal{K}, S, \beta) \models \Phi,\ \beta : \mathrm{VSym} \to \mathcal{D}\}$$

Concatenation of location descriptors is evaluated as union of their extensions.

[3] The attribute `length` returns the length for each array and is unspecified otherwise.

Note 2. This definition works for any kind of location descriptor. In particular, the guard formula Φ and possible subterms of the location term can be arbitrary ODL formulae and terms that may contain non-rigid symbols or even programs.

3.2 Syntax and Semantics of Symbols with Explicit Dependencies

The notation introduced above provides a concise way to characterise sets of locations. Now we extend the names of non-rigid (predicate and function) symbols by qualifications in the form of location descriptors. The idea is that the value of thus qualified symbols depends at most on the values of their arguments *plus those locations contained in the extension of their location descriptor.*

Definition 14 (Symbols with Explicit Dependencies). *Let ld denote a semicolon-separated list of location descriptors and let $p : T_1 \times \cdots \times T_n$ be a non-rigid predicate. The non-rigid predicate symbol $p[ld] : T_1 \times \cdots \times T_n$ is called* predicate with explicit dependencies*. Analogously for functions.*

The definitions of terms and formulae remain unchanged. The only difference is that the signature contains the above defined location-dependent symbols.

There are only few restrictions on the interpretation of ordinary non-rigid function and predicate symbols: essentially, their interpretation has to be well-defined and respect their types. The situation is different for symbols with explicit dependencies. The interpretation of a symbol $p[ld]$ with explicit dependencies has to coincide on all states S_1, S_2 that share the same values for the locations described by ld. This is precisely formulated in the next two definitions.

Definition 15 (ld-Equivalence). *For any Kripke structure $\mathcal{K}$ we define for any location descriptor ld the equivalence relation $\approx_{ld}$ on states where $S_1 \approx_{ld} S_2$ iff the following two conditions hold:*

1. *$ld^{(\mathcal{K},S_1)} = ld^{(\mathcal{K},S_2)}$ (identical location descriptor extension) and*
2. *$f^{(\mathcal{K},S_1)}(d_1,\ldots,d_n) = f^{(\mathcal{K},S_2)}(d_1,\ldots,d_n)$ for any $\langle f,(d_1,\ldots,d_n)\rangle \in ld^{(\mathcal{K},S_1)}$.*

The additional restriction required for Kripke structures to accomodate symbols with explicit dependencies is now covered by the definition:

Definition 16 (Dependency-Consistent Kripke Structure). *We call an ODL Kripke structure $\mathcal{K} := (\mathcal{M},\mathcal{S},\rho)$* dependency-consistent *if for any predicate $k[ld]$ (function $f[ld]$) depending on ld and any two states S_1, S_2 with $S_1 \approx_{ld} S_2$ the evaluation of predicate $k[ld]$ (function $f[ld]$) in S_1 and S_2 coincides.*

Lemma 1. *Let $\mathcal{K}$ be a dependency-consistent Kripke structure. Then in any two states S_1, $S_2 \in \mathcal{S}$ with $t_i^{(\mathcal{K},S_1)} = t_i^{(\mathcal{K},S_2)}$ $(i \in \{1\ldots n\})$ the atomic formula $p[ld](t_1,\ldots,t_n)$ evaluates to the same truth value whenever $S_1 \approx_{ld} S_2$ holds. Analogously for location dependent function symbols.*

Note 3. The notions of satisfiability, validity and model carry over to dependency-consistent Kripke structures in a straightforward manner. From now on we deal only with dependency-consistent Kripke structures and the corresponding notions of model, validity, etc., if not stated otherwise.

Example 3. Instead of modelling nonNullArray as a non-rigid predicate as done on p. 28, it can now be modelled as a predicate symbol with explicit dependencies: nonNullArray[`\for` $T[]\ a$, int i; $a[i]] : T[]$. This expresses explicitly that its value depends only on the value of the components of $T[]$-typed arrays and, of course, on its argument.

4 Correct Definitional Extensions

Definitional extension in first-order logic preserves consistency, i.e., adding a new axiom of the form $\forall \overline{x}(p(\overline{x}) \leftrightarrow \phi)$ for a new predicate symbol p not occurring in ϕ will not introduce new inconsistencies. For predicates with explicit dependencies an axiom of the form $\forall \overline{x} : \overline{T_{p[ld]}};\ (p[ld](\overline{x}) \leftrightarrow \phi)$ may already by inconsistent with respect to the dependency semantics of Def. 16: one has to ensure that the implicit state dependencies of ϕ are reflected in the location descriptor ld. The problem only concerns the implication $\forall \overline{x} : \overline{T_{p[ld]}};\ (p[ld](\overline{x}) \rightarrow \phi)$. We will consider only *implicational* axioms of this type and, to simplify the presentation, we assume there is only one axiom for each defined predicate. These axioms will be exploited during proof search as rewrite rules $p[ld](t_1, \ldots, t_n) \rightsquigarrow \phi(t_1, \ldots, t_n)$ named $\mathsf{axiom}_{p[ld]\rightarrow\phi}$, where the t_i's are terms of the appropriate type. In this setting we allow the defined symbol to occur recursively on the right-hand side.

We intend to define a proof obligation that, when valid, ensures a given implicational axiomatisation of a location-dependent symbol to be consistent with respect to dependency semantics. The problem of termination in the case of recursive rewrite rules is a different matter and has to be dealt with separately.

We need a new concept called *anonymising update for location descriptors*: this is a quantified update for a given location descriptor that assigns all locations in its extension a fixed, but unknown value:

Definition 17 (Anonymising Update for Location Descriptors). *Let* $ld := \texttt{\textbackslash for}\ \overline{T}\ \overline{o}; \texttt{\textbackslash if}\ (\phi)\ f(\overline{t})$ *be a location descriptor. The quantified update*

$$\mathcal{V}_{ld} := \texttt{\textbackslash for}\ \overline{T}\ \overline{o};\ \texttt{\textbackslash if}\ (\phi);\ f(\overline{t}) := c(\overline{t})$$

with c being a fresh uninterpreted rigid function symbol of matching type and arity is called an anonymising update *for the location descriptor ld. For $ld = ld_1; \ldots; ld_n$ we define $\mathcal{V}_{ld} = \mathcal{V}_{ld_1} \,||\, \ldots \,||\, \mathcal{V}_{ldn}$.*

With the help of anonymising updates it is possible to formulate dependence consistency (Def. 16) directly as the proof obligation

$$po_{p[ld]\rightarrow\phi} \quad := \quad \forall \overline{x} : \overline{T_{p[ld]}};\ ((\{\omega_c\}\ \{\mathcal{V}_{ld}\}\ \phi(\overline{x})) \leftrightarrow \{\mathcal{V}_{ld}\}\ \phi(\overline{x}))) \qquad (1)$$

where ω_c is a fresh anonymous program update (Def. 10).

Theorem 1. *If* (1) *is logically valid, then* $\forall \overline{x} : \overline{T_{p[ld]}};\ (p[ld](\overline{x}) \rightarrow \phi)$ *is consistent with respect to dependency semantics.*

5 Simplification Rules

In this section we introduce two update simplification rules that can be applied to arbitrary atomic formulas with explicit dependencies. We make use of the following Lemma [3, Sect. 3.9], [18]:

Lemma 2. *Two updates* $\mathcal{U}_1$, $\mathcal{U}_2$ *are called* equivalent *if they induce the same state transformer in any Kripke structure. Then every update* $\mathcal{U}$ *that has no anonymous program update as a component is equivalent to an update of the form* $upPart_1 \,||\, \ldots \,||\, upPart_{len}$ *with* $upPart_i := \backslash\texttt{for}\ \overline{T_i}\ \overline{x_i};\ \backslash\texttt{if}\ (\tau_i);\ g_i(\overline{u_i}) := val_i$.

Crucial for the first simplification rule is the notion of *relevant location symbol.* Relevant location symbols are a syntactic approximation of the location symbols that a formula ϕ or a term t may depend on. The notation is $Loc(\phi)$ and $Loc(t)$. In the definition below, and later, we use the following notation:

- $h[ld](s_1, \ldots, s_n)$ stands for a function or predicate symbol with explicit dependencies, $ld = ld_1; \ldots; ld_q$;
- $ld_i := \backslash\texttt{for}\ T_{i1}\ o_{i1}; \ldots; T_{ir_i}\ o_{ir_i}; \backslash\texttt{if}\ (\phi_i)\ f_i(t_{i1}, \ldots, t_{i\alpha_i})$

For ease of presentation we assume that $f_i \neq f_j$ for $i \neq j$. This affects the formula $\psi_{\mathcal{U}_1,\mathcal{U}_2,ld}$ in Def. 20. It is not hard to see how to adapt this formula to the unrestricted case.

Definition 18 (Relevant Location Symbols).

$Loc(\neg\psi)$	$:= Loc(\psi)$	
$Loc(\mathcal{Q}\ T\ x; \psi)$	$:= Loc(\psi)$	$\mathcal{Q} \in \{\forall, \exists\}$
$Loc(\psi \circ \phi)$	$:= Loc(\psi) \cup Loc(\phi)$,	$\circ \in \{\wedge, \vee, \rightarrow, \ldots\}$
$Loc(h(s_1, \ldots, s_n))$	$:= \bigcup_{i=1}^{n} Loc(s_i)$,	*h is a rigid symbol*
$Loc(\langle prg \rangle \psi)$	$:= \mathrm{FSym}_{loc}$	*see Def. 2*
$Loc([prg]\psi)$	$:= \mathrm{FSym}_{loc}$	*see Def. 2*
$Loc(g(s_1, \ldots, s_n))$	$:= \{g\} \cup \bigcup_{i=1}^{n} Loc(s_i)$	*g is a location fct. symbol*
$Loc(ld)$	$:= \bigcup_{i=1}^{q} \big(Loc(f_i(t_{i1}, \ldots, t_{i\alpha_i})) \cup Loc(\phi_i)\big)$	*ld as stipulated above*
$Loc(h[ld](s_1, \ldots, s_n))$	$:= Loc(ld) \cup \bigcup_{i=1}^{n} Loc(s_i)$	
$Loc(h(s_1, \ldots, s_n))$	$:= \mathrm{FSym}_{loc}$	*if h is a non-rigid symbol, but not a location symbol and without explicit dep.*
$Loc(\mathcal{U})$	$:= \bigcup_{i=1}^{len} (Loc(g_i(\overline{u_i})) \cup Loc(\tau_i) \cup Loc(val_i))$	$\mathcal{U}$ *as in Lemma 2*
$Loc(\omega_c)$	$:= \mathrm{FSym}_{loc}$	*see Defs. 2, 10*
$Loc(\{\mathcal{U}\}\xi)$	$:= Loc(\mathcal{U}) \cup Loc(\xi)$	ξ *a formula or term*

Lemma 3. *Let ϕ be an arbitrary formula, $\mathcal{K}$ a Kripke structure, S a state in $\mathcal{K}$, $\mathcal{U}$ as in Lemma 2 with $g_i \notin Loc(\phi)$ for all i then*

$$(\mathcal{K}, S) \models \phi \text{ iff } (\mathcal{K}, S) \models \{\mathcal{U}\}\phi$$

This lemma guarantees the correctness of the following rule.

Definition 19 (Coincidence Simplification Rule).

$$\{\mathcal{U}\}\ k[ld](s_1, \ldots, s_n) \rightsquigarrow \{\mathcal{U}'\}\ k[ld](s_1, \ldots, s_n)$$

where $\mathcal{U}' := \mathcal{U} - \{upPart_i \mid g_i \notin Loc(k[ld](s_1, \ldots, s_n))\}$ *and* $\mathcal{U}$ *as in Lemma 2.*

Example 4. Here is an instance of the coincidence simplification rule, where s is a term with $\mathtt{j} \notin Loc(s)$: $(\{\mathtt{i} := iVal \,||\, \mathtt{j} := jVal\}\ p[\mathtt{i}](s)) \rightsquigarrow (\{\mathtt{i} := iVal\}\ p[\mathtt{i}](s))$

Example 5. The coincidence simplification rule allows to prove the motivating example on p. 28 easily: in nonNullArray$(a) \rightarrow \{\mathtt{j} := \mathtt{j} + 1\}$ nonNullArray(a) we use nonNullArray[\for T[] a, int i; $a[i]$] as introduced in Example 3. For $\mathtt{j} \notin Loc(a)$ an instance of this rule simplifies

$$\{\mathtt{j} := \mathtt{j} + 1\}\ \text{nonNullArray}[\ldots](a) \rightsquigarrow \text{nonNullArray}[\ldots](a)$$

rendering the implication trivially valid.

The coincidence simplification rule is of an approximate nature but is sufficient for many practical purposes. We provide a stronger, semantic simplification rule in the form of the *equivalence simplification rule.* We require sufficient and necessary conditions for the logical equivalence of the two formulae $\{\mathcal{U}_1\}\ k[ld](s_1, \ldots, s_n)$ and $\{\mathcal{U}_2\}\ k[ld](s_1, \ldots, s_n)$. First we want the argument terms of $k[ld]$ to evaluate to the same values after the respective updates, i.e., $\{\mathcal{U}_1\}s_l \doteq \{\mathcal{U}_2\}s_l$ for all $1 \leq l \leq n$. Next we want to formalise that the locations described by each ld_j after update $\mathcal{U}_1$ are the same as those described by ld_j after update $\mathcal{U}_2$ and that their values coincide:

$$\begin{aligned}
&\psi_j^1 = \forall d; \forall \overline{c_j}; \forall \overline{o_j}; \\
&\quad \big\{(\{\mathcal{U}_1\}\ (\phi_j \wedge \overline{c_j} \doteq \overline{t_j} \wedge d \doteq \mathtt{f}_j(\overline{t_j}))) \rightarrow \exists \overline{o_j}(\{\mathcal{U}_2\}\ (\phi_j \wedge \overline{c_j} \doteq \overline{t_j} \wedge d \doteq \mathtt{f}_j(\overline{t_j})))\big\} \\
&\psi_j^2 = \forall d; \forall \overline{c_j}; \forall \overline{o_j}; \\
&\quad \big\{(\{\mathcal{U}_2\}\ (\phi_j \wedge \overline{c_j} \doteq \overline{t_j} \wedge d \doteq \mathtt{f}_j(\overline{t_j}))) \rightarrow \exists \overline{o_j}(\{\mathcal{U}_1\}\ (\phi_j \wedge \overline{c_j} \doteq \overline{t_j} \wedge d \doteq \mathtt{f}_j(\overline{t_j})))\big\}
\end{aligned}$$

It is now not hard to see that $\psi_{\mathcal{U}_1,\mathcal{U}_2,ld} = \bigwedge_{j=1}^{q}(\psi_j^1 \wedge \psi_j^2) \wedge \bigwedge_{l=1}^{n}\{\mathcal{U}_1\}s_l \doteq \{\mathcal{U}_2\}s_l$ is valid if and only if $\{\mathcal{U}_1\}\ k[ld](s_1, \ldots, s_n) \leftrightarrow \{\mathcal{U}_2\}\ k[ld](s_1, \ldots, s_n)$ is valid. This justifies the following rule:

Definition 20 (Equivalence Simplification Rule).

$\{\mathcal{U}_1\}\ k[ld](s_1, \ldots, s_n) \rightsquigarrow \{\mathcal{U}_2\}\ k[ld](s_1, \ldots, s_n)$ *provided formula* $\psi_{\mathcal{U}_1,\mathcal{U}_2,ld}$ *holds.*

Example 6. We modify Ex. 5 by using the more specialised 0-ary predicate $\mathrm{nonNullArray}_a[\texttt{\textbackslash for}\ \mathrm{int}\ i;\ a[i]]$ restricting the non-null element property to only those arrays referenced by the program variable a. Let

$$a \neq b \rightarrow (\mathrm{nonNullArray}_a[\ldots] \rightarrow \{b[0] := \mathrm{null}\}\ \mathrm{nonNullArray}_a[\ldots])$$

the formula to be proven valid where b is an array of the same type as a. Note that we cannot apply the coincidence simplification rule here: $\mathit{Loc}(\texttt{\textbackslash for}\ \mathrm{int}\ i;\ a[i])$ yields $[\cdot]$, i.e., the non-rigid array lookup function which is also the leading function symbol of the update $\{b[0] := \mathrm{null}\}$. Now we apply the equivalence simplification rule which gives for formula ψ_1^1:

$$\begin{array}{l}\forall d{:}T; c_1 : T[]; c_2{:}\mathrm{int}; i{:}\mathrm{int};\\ \quad ((\{b[0] := \mathrm{null}\}\ (\mathit{true} \land c_1 \doteq a \land c_2 \doteq i \land d \doteq a[i])) \rightarrow\\ \qquad\quad \exists i{:}\mathrm{int};\ (\mathit{true} \land c_1 \doteq a \land c_2 \doteq i \land d \doteq a[i]))\end{array}$$

Under the assumption $a \neq b$ this can be easily proven with a first-order theorem prover after applying the update.

Note 4. The calculus with the equivalence simplification rule added is complete for the logic containing symbols with explicit dependencies relative to the calculus without those symbols.

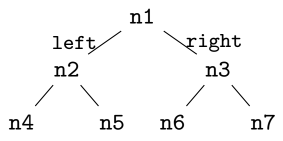
6 Applications and Case Studies

In this section we show applications of symbols with explicit dependency information modelling specification predicates such as reachability. Using these symbols provides advantages for interactive and automated proving. By being able to delay expanding the definition of a non-recursive predicate, the proof remains readable for a human reader. The automation benefits from the availability of general simplification rules reducing the need to look into the definition of non-rigid symbols. This is briefly illustrated in two case studies.

6.1 Specification predicates

Reachability. Treatment of linked data structures in specification and verification of object-oriented programs requires routinely to express reachability between two objects via a finite chain of fields. Fig. 1 shows a simple binary, directed tree, where subtrees are accessible via the fields `left` and `right`.

Specifying properties such as an element occurring in a given list requires to introduce and formalise the concept of reachability for these data structures. In the following we concentrate on the definition of a predicate formalising reachability for the directed, binary tree structure in Fig. 1. We aim to define a



Subtree at `n5` is reachable from the root `n1` via fields `left` and `right` in exactly 2 steps:
`reach[\for Tree` t; t`.left;\for Tree` t; t`.right](n1, n5, 2)`

Fig. 1. A binary tree datastructure with its associated reachable predicate symbol.

predicate that takes three arguments x, y (both of type `Tree`) and n (of type `int`) and holds if the subtree y can be reached from tree x in exactly n steps using only the fields `left` and `right`. The recursive definition for the reachability predicate is rather straightforward:

$$\mathtt{reach}(x,y,n) :\Leftrightarrow \begin{cases} \text{false} & ,\text{if } n < 0 \\ x \doteq y & ,\text{if } n \doteq 0 \\ (x\texttt{.left} \not\doteq \text{null} \wedge \mathtt{reach}(x\texttt{.left}, y, n-1)) \vee & \\ (x\texttt{.right} \not\doteq \text{null} \wedge \mathtt{reach}(x\texttt{.right}, y, n-1)) & ,\text{if } n > 0 \end{cases}$$

A calculus rule can be easily derived from the definition. The predicate is clearly non-rigid as it depends not only on the value of its arguments but also on the `left` and `right` fields of the subtrees below and including x. Therefore, we easily identify a location descriptor capturing all location dependencies, namely `\for Tree` t; t`.left`; `\for Tree` t; t`.right` describing the set of all `left`, `right` locations of all trees. Hence, the location-dependent predicate symbol for capturing reachability in our notation is:

$$\mathtt{reach}[\backslash\mathtt{for\ Tree}\ t;\ t.\mathtt{left};\backslash\mathtt{for\ Tree}\ t;\ t.\mathtt{right}](\mathtt{Tree},\mathtt{Tree},\text{int})\ .$$

6.2 Selection Sort

Verification of a sorting algorithm consists of showing that the result is indeed sorted and that the result contains exactly the elements of the original collection, in other words, that the result is a permutation of the input. For the verification of a standard selection sort implementation in Java a specification predicate with explicit dependencies has been used to formalise permutation of two arrays.[4]

The permutation predicate $\mathtt{perm}[\backslash\mathtt{for}\ \text{int}[]\ o;\text{int}\ i;\ o[i]](\text{int}[],\ \text{int}[])$ is defined with the help of a recursive count of the occurrences of all elements in both arrays. The predicate occurs once in the post condition stating that the resulting array is indeed a permutation of the original, and the second time in the loop invariant of the sorting algorithm.

The selection sort algorithm starts at the first array element and swaps it with the first minimal element encountered in the subsequent array continuing until

[4] This case study has been joint work and is also reported in [19], where the focus was an improved loop invariant rule.

the last element is reached. Therefore, it utilises two nested loops. To prove that both loops preserve the permutation property without predicates having explicit location dependencies requires several inductive subproofs. To substantiate this claim we look at the following formula[5] that occurs as subgoal during the proof and has to be proven valid:

$$\begin{aligned}
&\{\backslash\texttt{for}\ \text{int}\ i;\ aCopy[i] := a0[i]\,||\,\backslash\texttt{for}\ \text{int}\ i;\ a0[i] := get0(a,i)\}\ \texttt{perm}[\ldots](a0, aCopy) \rightarrow\\
&\quad \{b10 := FALSE\,||\,b4 := TRUE\,||\,b5 := FALSE\,||\,\ldots\,||\\
&\quad\ \backslash\texttt{for}\ \text{int}\ i;\ aCopy[i] := a0[i]\,||\,\backslash\texttt{for}\ \text{int}\ i;\ a0[i] := get0(a,i)\,||\\
&\quad\ a0[idx1] := get0(a0, idx2)\,||\,a0[idx2] := get0(a0, idx1)\}\ \texttt{perm}[\ldots](a0, aCopy)
\end{aligned}$$

The update of the formula in the implication's premise states a transition to a state where array *aCopy* is a shallow copy of the *old* content of array $a0$. *Simultaneously* new values are assigned to the components of array $a0$. The complete formula in the premise states then that in this state *aCopy* and $a0$ are permutations.

The update of the formula in the conclusion starts with a number of updates stemming from branch predicates irrelevant for the permutation property. The quantified updates assign the components of the arrays *aCopy* and $a0$ the exact same values as it is the case for the premise formula. *But* the following two updates lead to a slightly different state with $a0[idx1]$ and $a0[idx2]$ swapped.

Obviously, the implication is valid, because a standard transposition lemma can be used to show preservation of the permutation property. The technical difficulty is that a straightforward application of this transposition lemma is not possible. One has to prove that all updates preceding the array locations including the twelve updates abbreviated by "…" have no effect on the permutation property. This is done inductively using a strengthening of the claim where for any possible value of $b10$, $b4$, $b5, \ldots$ the permutation property remains unchanged.

In contrast, the approach presented in this paper exploiting dependency information allows to prove the formula valid without induction. Application of the *Coincidence Simplification Rule* (Def. 19) simplifies the state representation in the conclusion so far that the application of the transposition lemma is directly possible. We were able to prove the permutation property for an executable Java implementation of selection sort with only one user interaction exploiting the fact that swapping exactly two elements in an array preserves the permutation property.

6.3 Schorr-Waite Algorithm

The most complex case study so far where predicates with explicit dependencies were used is the verification of a fully functional Java implementation of the

[5] Slightly simplified and beautified.

Schorr-Waite graph marking algorithm [20] for arbitrary finite graph structures. The abstract algorithm has been verified in several case studies [7, 6, 21, 16, 1, 5, 12], mostly for the case of binary graphs. Our case study, first reported in [8], is to our knowledge the first time that an executable Java implementation for the general case was verified.

The Schorr-Waite graph marking algorithm saves memory by avoiding to encode the taken path in the method call stack. Instead a small number of auxiliary variables and subtle pointer rotations are used to encode the backtracking path. Besides showing that all reachable nodes are visited, the challenge is to show that afterwards the graph structure is restored, because the traversal uses destructive pointer manipulations.

Specification predicates with explicit dependencies were employed to express reachability of two nodes in the graph structure and for a specification predicate that characterises the backtracking path. The reachability predicate has been specified similar to the one in Sect. 6.1. The predicate characterising the backtracking path is specified in such a way that it evaluates to true if a given node is an element of the currently taken path. This is clearly state-dependent as the backtracking path changes during execution.

7 Related and Future Work

The Java Modelling Language (JML) [14] supports a depends clause used to express on which other fields a model field depends. The depends clause is then used to extend assignable clauses appropriately in case they contain model fields. These depends clauses have been introduced by Leino [15], the main idea being to replace the occurrence of an abstract/model field a with a function symbol $a'(f_1, \ldots, f_n)$ that explicitly enumerates the fields it depends on.

Separation logic [17] requires to specify exactly those locations of the heap (alternatively, to separate a heap into two orthogonal heaps) that are necessary to prove a property. The local judgements are then generalised by application of a frame rule. This approach is in the following sense complementary to ours: in separation logic, the shape of the heap is made explicit and dependencies are lifted with the help of a frame axiom. We make location dependencies explicit and can, therefore, abstract away from the concrete layout of the heap. This seems more appropriate for target languages such as Java. Another advantage of our approach is that only standard typed first-order logic is used.

Dynamic frames as introduced in [13] provide a uniform treatment of modified locations and dependencies, which are separate concerns in the presented work allowing for more precision. Of particular interest is the preservance operator Ξf for a dynamic frame (set of locations) f expressing that a computation does depend on or modify it. This property can be translated to our framework and logic. The other operators for dynamic frames deal with variants of modifies clauses in presence of object creation.

In [2] read effects are used for treating invocations of pure methods. Our approach is directly targeted at the level of specification predicates without the need to define them as pure boolean methods, it provides a compact and precise notion for the dependencies and works without an explicit heap presentation. Another concern of [2] is well-definedness of specifications containing pure methods. For our approach well-definedness is desirable, but will not cause unsoundness of the verification system.

Further related work focusses on data groups or similar concepts to support information hiding and encapsulation. This is ongoing work in the context of the KeY project.

Future work includes to exploit further application scenarios for predicates with explicit dependencies. Rümmer suggested to use them to achieve second-order like specification capabilities in a first-order dynamic logic through parameterisation with functions. This allows, for example, to use the same specification for the sum over all elements of an array $\sum_i f(\mathtt{a[i]})$ for any function f. Further, we will investigate how to improve automation of the equivalence update simplification rule, i.e., in finding a suitable equivalent update $\mathcal{U}_2$.

8 Conclusion

We introduced non-rigid specification predicate (and function) symbols that explicitly list the set of locations their value may depend on. Then we presented a verification framework for such symbols with explicit dependencies. This framework consists of two parts: on the one hand a uniformly generated proof obligation that ensures correctness of a predicate with explicit dependencies with respect to its axiomatisation; on the other hand a number of general simplification rules that allow one to exploit explicit dependencies within the context of a proof. Several application scenarios common in program verification as well as two case studies support the usefulness of our approach. The implementation and case studies were done within the KeY-system [3], however, as pointed out in the introduction the problem of location-dependent predicates as well as the solution presented here can be transferred to other program logics based on weakest precondition reasoning.

Acknowledgments

We thank Philipp Rümmer and Wolfgang Ahrendt for valuable comments. Special thanks go also to Benjamin Weiß for valuable comments and in particular for pointing out a problem in the original version of Def. 8. We thank the anonymous referees for their valuable comments.

References

1. J. Abrial. Event based sequential program development: Application to constructing a pointer program. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Proc. Formal Methods*, volume 2805 of *LNCS*, pages 51–74. Springer, September 2003.
2. Ádám Darvas and K. R. M. Leino. Practical reasoning about invocations and implementations of pure methods. In M. B. Dwyer and A. Lopes, editors, *Fundamental Approaches to Software Engineering (FASE)*, volume 4422 of *LNCS*, pages 336–351. Springer-Verlag, 2007.
3. B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer-Verlag, 2006.
4. B. Beckert and A. Platzer. Dynamic logic with non-rigid functions: A basis for object-oriented program verification. In U. Furbach and N. Shankar, editors, *Proc. Intl. Joint Conference on Automated Reasoning, Seattle, USA*, volume 4130 of *LNCS*, pages 266–280. Springer-Verlag, 2006.
5. L. Birkedal, N. Torp-Smith, and J. Reynolds. Local reasoning about a copying garbage collector. In *Proc. 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press, Jan. 2004.
6. R. Bornat. Proving pointer programs in Hoare logic. In R. C. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *LNCS*, pages 102–126. Springer, 2000.
7. M. Broy and P. Pepper. Combining algebraic and algorithmic reasoning: An approach to the Schorr-Waite algorithm. *ACM Trans. Program. Lang. Syst.*, 4(3):362–381, 1982.
8. R. Bubel. *Formal Verification of Recursive Predicates*. PhD thesis, Fakultät für Informatik, Univ. Karlsruhe, June 2007.
9. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
10. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. Foundations of Computing. MIT Press, Oct. 2000.
11. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, Oct. 1969.
12. T. Hubert and C. Marché. A case study of C source code verification: the Schorr-Waite algorithm. In B. K. Aichernig and B. Beckert, editors, *Proc. 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pages 190–199. IEEE Press, 2005.
13. I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2006.
14. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin. *JML Reference Manual*, Feb. 2007. Draft revision 1.200.
15. K. R. M. Leino. *Toward Reliable Modular Programs*. PhD thesis, Caltech, 1995.
16. F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. In F. Baader, editor, *Automated Deduction — CADE-19*, volume 2741 of *LNCS*, pages 121–135. Springer, 2003.
17. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.
18. P. Rümmer. Sequential, parallel, and quantified updates of first-order structures. In M. Hermann and A. Voronkov, editors, *Proc. Logic for Programming, Artificial Intelligence and Reasoning, Phnom Penh, Cambodia*, volume 4246 of *LNCS*, pages 422–436. Springer-Verlag, 2006.
19. S. Schlager. *Symbolic Execution as a Framework for Deductive Verification of Object-Oriented Programs*. PhD thesis, Fakultät für Informatik, Univ. Karlsruhe, Feb. 2007.
20. H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM*, 10(8):501–506, 1967.
21. H. Yang. Verification of the schorr-waite graph marking algorithm by refinement, 2003. Unpublished, http://ropas.kaist.ac.kr/ hyang/paper/dagstuhl-SW.pdf.