A verified VCGen based on Dynamic Logic: an exercise in meta-verification with Why3

Maria João Frade[a], Jorge Sousa Pinto[a]

[a] *HASLab/INESC TEC & Universidade do Minho, Portugal*

Abstract

With the incresasing importance of program verification, an issue that has been receiving more attention is the certification of verification tools, addressing the vernacular question: "Who verifies the verifier?". In this paper we approach this meta-verification problem by focusing on a fundamental component of program verifiers: the "Verification Conditions Generator" (VCGen), responsible for producing a set of proof obligations from a program and a specification. The semantic foundations of VCGens lie in program logics, such as Hoare logic, Dynamic logic, or Separation logic, and related predicate transformers.

Dynamic logic is the basis of the KeY system, one of the foremost deductive verifiers, whose logic makes use of the notion of *update*, which is quite intricate to formalize. In this paper we derive systematically, based on a KeY-style dynamic logic, a correct-by-construction VCGen for a toy programming language. Our workflow covers the entire process, from the logic to the VCGen. It is implemented in the Why3 tool, which is itself a program verifier. We prove the soundness and (an appropriate notion of) completeness of the logic, then define a VCGen for our language and establish its soundness.

Dynamic logic is one of a variety of research topics that our dear friend and colleague Luís Soares Barbosa has, over the years, initiated and promoted at the University of Minho. It is a pleasure for us to dedicate this work to him on the occasion of his 60th birthday.

Keywords: Hoare logic, Verification Conditions, Program Verification, Program Annotations, Weakest Preconditions, Updates

1. Introduction

KeY [1] is one of the major program verification tools that have, in the last 15 years, made deductive methods, based on the use of program logics, available to a wide range of users. One distinct aspect of it is that, while most other tools are based on Hoare logic [2] and the related weakest precondition calculus of

Email addresses: mjf@di.uminho.pt (Maria João Frade), jsp@di.uminho.pt (Jorge Sousa Pinto)

Dijkstra [3], or else on the more recent Separation logic [4], the underlying logic of KeY is *dynamic logic* [5, 6].

Dynamic logic includes *modalities*. The logic we consider in this paper uses the 'box' modality, which takes a program as argument. The formula $[C]\phi$ has the following meaning: "every terminating execution of C results in a state that satisfies ϕ" [5]. The properties involving this modality look extremely familiar from the standpoint of the weakest precondition calculus and Hoare logic:

$$[x := e]\phi \;=\; \phi[e/x] \qquad\qquad [C_1; C_2]\phi \;=\; [C_1][C_2]\phi$$

$$[\mathit{if}\ b\ \mathit{then}\ C_1\ \mathit{else}\ C_2]\phi \;=\; (b \to [C_1]\phi) \wedge (\neg b \to [C_2]\phi)$$

$$\frac{\theta \wedge b \;\to\; [C]\ \theta}{\theta \;\to\; [\mathit{while}\ b\ \mathit{do}\ C]\ (\theta \wedge \neg b)}$$

Initial work on dynamic logic had a theoretical focus on the problems created by extending first-order logic with program formulas, including the decidability of validity and completeness. A strong point of dynamic logic is that formulas may contain more than one program, allowing for properties to be expressed, such as program equivalence, that cannot be expressed in Hoare logic.

The notion of a state *update* was later introduced [7] in the context of work on the KeY system. The paper studied a dynamic logic for object-oriented programs that can be seen as the core of JavaDL, a program logic for Java programs. Updates are programs of a special form, essentially parallel assignments in which the same variable (or other mutable entities) may occur as left-hand side more than once. Application of an update to an expression or formula is just (parallel) variable substitution, with a "rightmost wins" strategy whenever a variable is assigned more than once. The system included rules that transform update modalities into applications of updates to formulas:

$$\frac{\mathcal{U}\ (\psi)}{[\mathcal{U}]\ \psi} \qquad\qquad \frac{[\mathcal{U}]([\mathcal{U}']\ \psi)}{[\mathcal{U};\ \mathcal{U}']\ \psi}$$

Crucially, a *simplification rule* is required to handle formulas like $[\mathcal{U}]([\mathcal{U}']\ \psi)$:

$$[\mathcal{U}]\ ([x_1 := e_1\ ||\ \ldots\ ||\ x_n := e_n]\ \psi) \;\rightsquigarrow\; [\mathcal{U}\ ||\ x_1 := \mathcal{U}\ (e_1)\ ||\ \ldots\ ||\ x_n := \mathcal{U}\ (e_n)]\ \psi$$

Consider for example the formula $x = K \to [x := 2{*}x;\ y := x]\ (x = 2{*}K \wedge y = x)$
Applying the rule shown above on the right will produce
$x = K \to [x := 2 * x]\ ([y := x]\ (x = 2 * K \wedge y = x))$
which can in turn be simplified as follows

- $x = K \to [x := 2 * x\ ||\ y := 2 * x]\ (x = 2 * K \wedge y = x)$
- $x = K \;\to\; (x = 2 * K \wedge y = x)\ \ [2 * x/x,\ 2 * x/y]$
- $x = K \;\to\; 2 * x = 2 * K \wedge 2 * x = 2 * x$

In later work on JavaDL by the KeY team [8], updates are no longer seen as programs, but rather as separate entities that can be applied to expressions, formulas, and other updates. The inference system deals specifically with formulas of the form $\phi \to \{\mathcal{U}\}[C]\psi$, where the consequent is constructed by applying an update to a formula containing a program (box modality).

Updates were introduced as a device to handle object aliasing (which we do not consider in the present paper), but they additionally allow for the introduction of a system with emphasis on *symbolic execution*: a forward view of formula propagation which, unlike the *strongest postcondition* calculus, does not introduce existential quantifiers. The system contains a specific set of rules for the control-flow constructs. In particular the sequence rule is split into several different rules according to the first statement, for instance:

$$\frac{\phi \ \wedge \{\mathcal{U}\}b \ \to \ \{\mathcal{U}\}[C_1 \,;\, C]\psi \qquad \phi \ \wedge \{\mathcal{U}\}(\neg b) \ \to \ \{\mathcal{U}\}[C_2 \,;\, C]\psi}{\phi \ \to \ \{\mathcal{U}\}[(\mathit{if}\ \ b\ \mathit{then}\ \ C_1\ \ \mathit{else}\ \ C_2) \,;\, C]\psi}$$

In this paper we consider a dynamic logic that we call WhileDL, which is essentially a fragment of JavaDL for while programs (so we focus on the "symbolic execution" aspect of JavaDL, rather than on object aliasing). Using the Why3 tool [9], we do the following: (i) we define the syntax and semantics of WhileDL and prove basic properties of the logic with the help of Why3 proof transformations and (mostly) external SMT solvers; (ii) we formalize an inference system (a 'calculus') for this logic, and mechanically prove its soundness and (a notion of) completeness; (iii) we introduce a Verification Conditions Generator (VCGen) that produces proof obligations in first-order logic, and prove its soundness with respect to the calculus.

The VCGen is a novel contribution. We remark that, unlike other major verifiers, KeY does not resort to external tools (in particular SMT solvers) to discharge proof obligations; instead, it includes an internal proof engine, which incorporates an update simplification mechanism. As will be seen in detail below, update simplification is specified in the KeY documentation as a set of rewrite rules, but no specific strategy is given for their application. Our VCGen includes one such strategy; the overall result is that we show how dynamic logic (and the update mechanism) can serve as the basis for an alternative verifier based on the use of external first-order provers.

In addition to the previous point, the paper serves as a case study in meta-verification, establishing a basis for VCGen certification using a deductive verifier. It highlights distinctive aspects of Why3, in particular the rich relationship between its logic and programming languages. The WhileDL VCGen is written as a program in WhyML (the internal Why3 programming language), and can be extracted as an OCaml program. With its specific logic and programming languages and its ability to interface with external proof tools, Why3 stands in a sweet spot, in terms of expressivity of logic language, degree of automation, and proof management, for the purpose of formalizing the different notions involved in this task. In particular, we reason about the (dynamic) program logic in Why3's logic language, but then use the programming language to actually implement the VCGen.

The paper is organized as follows: Section 2 introduces the syntax and semantics of WhileDL, including updates and the update simplification mechanism. Section 3 briefly presents the basics of the Why3 verifier. Sections 4, 5, and 6 contain the Why3 formalization, presenting respectively the logic, update simplification mechanism, and VCGen. Section 7 concludes the paper.

2. The WhileDL Logic

Syntax. WhileDL extends classical logic with two artifacts: a modal operator with programs, and updates. Both programs and updates denote state changes, but updates are simpler (in particular, they always terminate). We consider a simple While language with data types for integer numbers and Booleans. Statements include a do-nothing command, assignment, composition, (two-branched) conditional execution and while loop annotated with a loop invariant. The language has two base types `Int` and `Bool`. Arithmetic expressions are formed from constants and a set of variables $x \in \mathbf{Var}$, together with operators on integers.

$$
\begin{array}{llll}
\textbf{AExp} & a & ::= & \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \mid x \mid\ a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid \{\mathcal{U}\}\, a \\
\textbf{Upd} & \mathcal{U} & ::= & \texttt{skip} \mid x := a \mid \mathcal{U}_1 \| \mathcal{U}_2 \mid \{\mathcal{U}_1\}\, \mathcal{U}_2 \\
\textbf{BExp} & b & ::= & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 < a_2 \mid a_1 \leq a_2 \mid a_1 > a_2 \mid a_1 \geq a_2 \mid \\
 & & & \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \{\mathcal{U}\}\, b \\
\textbf{Form} & \phi & ::= & b \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \to \phi_2 \mid \forall x.\, \phi \mid \exists x.\, \phi \mid [C]\, \phi \mid \{\mathcal{U}\}\, \phi \\
\textbf{Stmt} & C & ::= & \texttt{skip} \mid x := a \mid\ C_1 \,;\, C_2 \mid \texttt{if}\ b\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \mid \texttt{while}\ b\ \texttt{do}\ \{\phi\}\, C
\end{array}
$$

In addition to aritmetic expressions $a \in \mathbf{AExp}$ and Boolean expressions $b \in \mathbf{BExp}$, we have formulas $\phi \in \mathbf{Form}$ that express properties of particular states of the programs. Our formulas are formulas of a first-order language (obtained as an expansion of Boolean expressions) with two new constructions: $[C]\, \phi$ and $\{\mathcal{U}\}\, \phi$. The formula $[C]\, \phi$ expresses the fact that every terminating execution of C leads to a state that satisfies ϕ. On the other hand, $\{\mathcal{U}\}\, \phi$ denotes the application of update $\mathcal{U}$ to formula ϕ (but note that updates can be applied to other entities, including expressions and even other updates).

Updates denote state changes. `skip` denotes the *empty update* that does not change anything. $x := a$ is an *elementary update* that assigns the value of the expression a to the variable x. $\mathcal{U}_1 \| \mathcal{U}_2$ denotes a *parallel update* that executes the updates $\mathcal{U}_1$ and $\mathcal{U}_2$ in parallel with a *last-win semantics*: if $\mathcal{U}_1$ and $\mathcal{U}_2$ attempt to assign conflicting values to a variable, then the value written by $\mathcal{U}_2$ prevails. $\{\mathcal{U}_1\}\, a$ (resp. $\{\mathcal{U}_1\}\, b$, $\{\mathcal{U}_1\}\, \phi$ and $\{\mathcal{U}_1\}\, \mathcal{U}_2$) denotes the *update aplication* whose meaning is that a (resp. b, ϕ and $\mathcal{U}_2$) will be evaluated in the state produced by the update $\mathcal{U}_1$.

The usual notion of *free variable* is extended to include updates and formulas of the form $[C]\, \phi$ in the following way:

$$
\begin{array}{lll}
\mathsf{FV}(\texttt{skip}) = \emptyset & & \mathsf{FV}([C]\, \phi) = \mathsf{FV}(\phi) \\
\mathsf{FV}(x := a) = \mathsf{FV}(a) & & \\
\mathsf{FV}(\mathcal{U}_1 \| \mathcal{U}_2) = \mathsf{FV}(\mathcal{U}_1) \cup \mathsf{FV}(\mathcal{U}_2) & & \\
\mathsf{FV}(\{\mathcal{U}\}\, t) = \mathsf{FV}(\mathcal{U}) \cup \mathsf{FV}(t) & \text{where} & t \in \mathbf{AExp} \cup \mathbf{BExp} \cup \mathbf{Form} \cup \mathbf{Upd}
\end{array}
$$

1. $\langle \texttt{skip}, s\rangle \Downarrow s$
2. $\langle x := a, s\rangle \Downarrow s[x \mapsto [\![a]\!](s)]$
3. if $\langle C_1, s\rangle \Downarrow s'$ and $\langle C_2, s'\rangle \Downarrow s''$, then $\langle C_1 \,;\, C_2, s\rangle \Downarrow s''$
4. if $[\![b]\!](s) = \mathbf{T}$ and $\langle C_1, s\rangle \Downarrow s'$, then $\langle \texttt{if}\ b\ \texttt{then}\ C_1\ \texttt{else}\ C_2, s\rangle \Downarrow s'$
5. if $[\![b]\!](s) = \mathbf{F}$ and $\langle C_2, s\rangle \Downarrow s'$, then $\langle \texttt{if}\ b\ \texttt{then}\ C_1\ \texttt{else}\ C_2, s\rangle \Downarrow s'$
6. if $[\![b]\!](s) = \mathbf{T}$, $\langle C, s\rangle \Downarrow s'$ and $\langle \texttt{while}\ b\ \texttt{do}\ \{\phi\}\, C, s'\rangle \Downarrow s''$, then $\langle \texttt{while}\ b\ \texttt{do}\ \{\phi\}\, C, s\rangle \Downarrow s''$
7. if $[\![b]\!](s) = \mathbf{F}$, then $\langle \texttt{while}\ b\ \texttt{do}\ \{\phi\}\, C, s\rangle \Downarrow s$

Figure 1: Natural semantics for While programs.

Semantics. Integer and Boolean expressions are interpreted as integer or Boolean values that depend on the values of variables that may occur in the expressions. In other words, they depend on a *state*, a function that maps each variable into an integer value. We will write $\Sigma = \mathbf{Var} \to \mathbb{Z}$ for the set of states. For $s \in \Sigma$, $x \in \mathbf{Var}$ and $z \in \mathbb{Z}$, $s[x \mapsto z]$ will denote the state that maps x to z and any other variable y to $s(y)$. The interpretation of $a \in \mathbf{AExp}$ and $b \in \mathbf{BExp}$ will be given by functions $[\![a]\!] : \Sigma \to \mathbb{Z}$ and $[\![b]\!] : \Sigma \to \{\mathbf{F}, \mathbf{T}\}$ respectively, reflecting our assumptions that an expression has a value at every state (evaluation always terminates without error) and expression evaluation never changes the state.

For the interpretation of formulas we take the usual interpretation of first-order formulas, with additional cases for formulas of the form $[C]\,\phi$ and $\{\mathcal{U}\}\,\phi$ (presented below). Note that, since formulas build on the language of program expressions their interpretation also depends on states, and states from Σ can be used as *variable assignments* in the interpretation of formulas. The interpretation of a formula $\phi \in \mathbf{Form}$ is then given by $[\![\phi]\!] : \Sigma \to \{\mathbf{F}, \mathbf{T}\}$.

For statements we consider a standard operational, natural style semantics, based on a deterministic *evaluation relation* $\Downarrow \subseteq \mathbf{Stmt} \times \Sigma \times \Sigma$ (building on an implicit interpretation of expressions). We will write $\langle C, s\rangle \Downarrow s'$ to denote the fact that if C is executed in the initial state s, then its execution terminates in the final state s'. The usual inductive definition of this relation is given in Figure 1. Note that invariant annotations do not affect the operational semantics.

The semantics of updates also depends on states: the interpretation of an update $\mathcal{U} \in \mathbf{Upd}$ in a given state is a function from Σ to Σ, $[\![\mathcal{U}]\!] : \Sigma \to (\Sigma \to \Sigma)$ defined as follows:

$$\begin{array}{lcl} [\![\texttt{skip}]\!](s)(s') & = & s' \\ [\![x := a]\!](s)(s') & = & s'[x \mapsto [\![a]\!](s)] \\ [\![\mathcal{U}_1 \| \mathcal{U}_2]\!](s)(s') & = & [\![\mathcal{U}_2]\!](s)([\![\mathcal{U}_1]\!](s)(s')) \\ [\![\{\mathcal{U}_1\}\,\mathcal{U}_2]\!](s)(s') & = & [\![\mathcal{U}_2]\!]([\![\mathcal{U}_1]\!](s)(s))(s') \end{array}$$

Observe that the first argument s is the state where the expressions that occur in the update are evaluated. Therefore, $[\![\mathcal{U}]\!](s)$ is a state transformation function, and $[\![\mathcal{U}]\!](s)(s')$ is the state produced by this function when applied to state s'.

Expressions are interpreted in the usual way. We state here only the case of

1. $\{...\|x := a_1\|...\|x := a_2\|...\}\, t \rightsquigarrow \{...\|\texttt{skip}\|...\|x := a_2\|...\}\, t$ where $t \in \mathbf{AExp} \cup \mathbf{BExp} \cup \mathbf{Form} \cup \mathbf{Upd}$
2. $\{...\|x := a\|...\}\, t \rightsquigarrow \{...\|\texttt{skip}\|...\}\, t$ where $t \in \mathbf{AExp} \cup \mathbf{BExp} \cup \mathbf{Form} \cup \mathbf{Upd}$ and $x \notin \mathsf{FV}(t)$
3. $\{\mathcal{U}_1\}\,\{\mathcal{U}_2\}\, t \rightsquigarrow \{\mathcal{U}_1\|\,\{\mathcal{U}_1\}\,\mathcal{U}_2\}\, t$ where $t \in \mathbf{AExp} \cup \mathbf{BExp} \cup \mathbf{Form} \cup \mathbf{Upd}$
4. $\{\mathcal{U}\|\,\texttt{skip}\}\, t \rightsquigarrow \{\mathcal{U}\}\, t$ where $t \in \mathbf{AExp} \cup \mathbf{BExp} \cup \mathbf{Form} \cup \mathbf{Upd}$
5. $\{\texttt{skip}\|\,\mathcal{U}\}\, t \rightsquigarrow \{\mathcal{U}\}\, t$ where $t \in \mathbf{AExp} \cup \mathbf{BExp} \cup \mathbf{Form} \cup \mathbf{Upd}$
6. $\{\texttt{skip}\}\, t \rightsquigarrow t$ where $t \in \mathbf{AExp} \cup \mathbf{BExp} \cup \mathbf{Form} \cup \mathbf{Upd}$
7. $\{\mathcal{U}\}\, t \rightsquigarrow t$ where $t \in \mathbf{Var} \cup \{\texttt{true}, \texttt{false}\} \cup \{..., -1, 0, 1, ...\}$
8. $\{\mathcal{U}\}\,(a_1 \bullet a_2) \rightsquigarrow (\{\mathcal{U}\}\, a_1) \bullet (\{\mathcal{U}\}\, a_2)$ where $\bullet \in \{+, *, -, =, <, >, \leq, \geq\}$
9. $\{\mathcal{U}\}\,\neg b \rightsquigarrow \neg\{\mathcal{U}\}\, b$
10. $\{\mathcal{U}\}\,(b_1 \bullet b_2) \rightsquigarrow (\{\mathcal{U}\}\, b_1) \bullet (\{\mathcal{U}\}\, b_2)$ where $\bullet \in \{\wedge, \vee\}$
11. $\{\mathcal{U}\}\,\neg\phi \rightsquigarrow \neg\{\mathcal{U}\}\,\phi$
12. $\{\mathcal{U}\}\,(\phi_1 \bullet \phi_2) \rightsquigarrow (\{\mathcal{U}\}\,\phi_1) \bullet (\{\mathcal{U}\}\,\phi_2)$ where $\bullet \in \{\wedge, \vee, \rightarrow\}$
13. $\{\mathcal{U}\}\,\forall x.\,\phi \rightsquigarrow \forall x.\,\{\mathcal{U}\}\,\phi$ where $x \notin \mathsf{FV}(\mathcal{U})$
14. $\{\mathcal{U}\}\,\exists x.\,\phi \rightsquigarrow \exists x.\,\{\mathcal{U}\}\,\phi$ where $x \notin \mathsf{FV}(\mathcal{U})$
15. $\{\mathcal{U}\}\,(x := a) \rightsquigarrow x := \{\mathcal{U}\}\, a$
16. $\{\mathcal{U}\}\,\texttt{skip} \rightsquigarrow \texttt{skip}$
17. $\{\mathcal{U}\}\,(\mathcal{U}_1\|\,\mathcal{U}_2) \rightsquigarrow (\{\mathcal{U}\}\,\mathcal{U}_1)\|\,(\{\mathcal{U}\}\,\mathcal{U}_2)$
18. $\{x := a\}\, x \rightsquigarrow a$

Figure 2: Simplification rules for updates.

update aplications:

$$\begin{array}{rcl} [\![\{\mathcal{U}\}\, a]\!](s) & = & [\![a]\!]([\![\mathcal{U}]\!](s)(s)) \\ [\![\{\mathcal{U}\}\, b]\!](s) & = & [\![b]\!]([\![\mathcal{U}]\!](s)(s)) \end{array}$$

For the interpretation of formulas we take the usual interpretation of first-order formulas extended with the two following cases:

$$\begin{array}{lll} [\![\{\mathcal{U}\}\,\phi]\!](s) = \mathbf{T} & \text{iff} & [\![\phi]\!]([\![\mathcal{U}]\!](s)(s)) = \mathbf{T} \\ [\![[C]\,\phi]\!](s) = \mathbf{T} & \text{iff} & [\![\phi]\!](s') = \mathbf{T} \ \text{ for all } s' \text{ such that } \ \langle C, s\rangle \Downarrow s' \end{array}$$

The remaining cases are interpreted as expected.

The intrinsic features and last-win semantics of updates make it possible to perform some simplifications on them that preserve the semantics. Figure 2 shows a set of simplification rules for update application, following JavaDL [8]. Observe that, as parallel composition is associative, $\mathcal{U}_1\|\,\mathcal{U}_2\|\,\mathcal{U}_3$ can be written instead of $\mathcal{U}_1\|\,(\mathcal{U}_2\|\,\mathcal{U}_3)$ or $(\mathcal{U}_1\|\,\mathcal{U}_2)\|\,\mathcal{U}_3$. The only case not covered by the simplification rules is that of $\{\mathcal{U}\}\,[C]\,\phi$. In this case the statement C must be first eliminated using the symbolic execution rules.

Calculus. The JavaDL calculus underlying the KeY tool is an inference system covering the full spectrum of formulas of the logic (including FOL). Our approach in this paper is quite different, and has more resemblances with the system presented in [10]. Our goal is to produce a verification conditions generator that will produce a set of first-order proof obligations (free of the modalities

$$\frac{\phi \to \{\mathcal{U}\}\,\psi}{\phi \Longrightarrow \{\mathcal{U}\}\,[\texttt{skip}]\,\psi} \quad \text{(skip)}$$

$$\frac{\phi \to \{\mathcal{U}\}\,(\{x := a\}\,\psi)}{\phi \Longrightarrow \{\mathcal{U}\}\,[x := a]\,\psi} \quad \text{(assign)}$$

$$\frac{\phi \wedge \{\mathcal{U}\}\,b \Longrightarrow \{\mathcal{U}\}\,[C_1]\,\psi \quad \phi \wedge \{\mathcal{U}\}\,\neg b \Longrightarrow \{\mathcal{U}\}\,[C_2]\,\psi}{\phi \Longrightarrow \{\mathcal{U}\}\,[\texttt{if}\ b\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\,\psi} \quad \text{(if)}$$

$$\frac{\phi \to \{\mathcal{U}\}\,\theta \quad \theta \wedge b \Longrightarrow \{\texttt{skip}\}\,[C]\,\theta \quad \theta \wedge \neg b \to \psi}{\phi \Longrightarrow \{\mathcal{U}\}\,[\texttt{while}\ b\ \texttt{do}\ \{\theta\}\ C]\,\psi} \quad \text{(while)}$$

$$\frac{\phi \Longrightarrow \{\mathcal{U}\}\,[C]\,\psi}{\phi \Longrightarrow \{\mathcal{U}\}\,[\texttt{skip}\,;\,C]\,\psi} \quad \text{(skip-seq)}$$

$$\frac{\phi \Longrightarrow \{\mathcal{U} \| \{\mathcal{U}\}\,(x := a)\}\,[C]\,\phi}{\phi \Longrightarrow \{\mathcal{U}\}\,[x := a\,;\,C]\,\phi} \quad \text{(assign-seq)}$$

$$\frac{\phi \wedge \{\mathcal{U}\}\,b \Longrightarrow \{\mathcal{U}\}\,[C_1\,;\,C_3]\,\psi \quad \phi \wedge \{\mathcal{U}\}\,\neg b \Longrightarrow \{\mathcal{U}\}\,[C_2\,;\,C_3]\,\psi}{\phi \Longrightarrow \{\mathcal{U}\}\,[(\texttt{if}\ b\ \texttt{then}\ C_1\ \texttt{else}\ C_2)\,;\,C_3]\,\psi} \quad \text{(if-seq)}$$

$$\frac{\phi \to \{\mathcal{U}\}\,\theta \quad \theta \wedge b \Longrightarrow \{\texttt{skip}\}\,[C_1]\,\theta \quad \theta \wedge \neg b \Longrightarrow \{\texttt{skip}\}\,[C_2]\,\psi}{\phi \Longrightarrow \{\mathcal{U}\}\,[(\texttt{while}\ b\ \texttt{do}\ \{\theta\}\ C_1)\,;\,C_2]\,\psi} \quad \text{(while-seq)}$$

$$\frac{\phi \Longrightarrow \{\mathcal{U}\}\,[C_1\,;\,(C_2\,;\,C_3)]\,\psi}{\phi \Longrightarrow \{\mathcal{U}\}\,[(C_1\,;\,C_2)\,;\,C_3]\,\psi} \quad \text{(seq-seq)}$$

Figure 3: Symbolic execution calculus for WhileDL update triples.

and updates that are present in WhileDL). For this purpose we introduce a calculus that manipulates a specific form of judgment (not WhileDL formulas). Similarly to Hoare logic, whose inference system contains rules that have FOL formulas as premises (or side conditions, whose validity checking is externalized, not included in the system), some rules in our calculus have WhileDL formulas as premises that cannot be derived within the calculus.

We will consider judgements of the form $\phi \Longrightarrow \{\mathcal{U}\}\,[C]\,\psi$ which we will call *update triples*. An update triple $\phi \Longrightarrow \{\mathcal{U}\}\,[C]\,\psi$ is *valid* iff for every $s \in \Sigma$, $[\![\phi \to \{\mathcal{U}\}\,[C]\,\psi]\!](s) = \mathbf{T}$. Figure 3 presents a set of inference rules for update triples, which play in this system a similar role to Hoare triples in Hoare logic. This system can be seen as a symbolic execution calculus that follows the program structure in a forward way, and produces WhileDL formulas as proof obligations when atomic programs and while loops are reached.

As an example, we show the derivation tree for a program that swaps the

values of variables x and y. Bottom-up we sequentially apply rules (seq-seq), (assign-seq), (assign-seq) and (assign).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(x = a \wedge y = b) \to \{\texttt{skip}\|\ \{\texttt{skip}\}\ (x := x + y)\|\ \{\texttt{skip}\|\ \{\texttt{skip}\}\ (x := x + y)\}\ y := x - y\}\ (\{x := x - y\}\ (x = b \wedge y = a))}{(x = a \wedge y = b) \Longrightarrow \{\texttt{skip}\|\ \{\texttt{skip}\}\ (x := x + y)\|\ \{\texttt{skip}\|\ \{\texttt{skip}\}\ (x := x + y)\}\ y := x - y\}\ [x := x - y]\ (x = b \wedge y = a)}}{(x = a \wedge y = b) \Longrightarrow \{\texttt{skip}\|\ \{\texttt{skip}\}\ (x := x + y)\}\ [y := x - y\ ;\ x := x - y]\ (x = b \wedge y = a)}}{(x = a \wedge y = b) \Longrightarrow \{\texttt{skip}\}\ [x := x + y\ ;\ (y := x - y\ ;\ x := x - y)]\ (x = b \wedge y = a)}}{(x = a \wedge y = b) \Longrightarrow \{\texttt{skip}\}\ [(x := x + y\ ;\ y := x - y)\ ;\ x := x - y]\ (x = b \wedge y = a)}$$

Applying the simplification rules for updates, we get

$$\begin{array}{l}
(x = a \wedge y = b) \to \{\texttt{skip}\|\ \{\texttt{skip}\}\ (x := x + y)\|\ \{\texttt{skip}\|\ \{\texttt{skip}\}\ (x := x + y)\}\ y := x - y\}\ (\{x := x - y\}\ (x = b \wedge y = a)) \\
\rightsquigarrow^* (x = a \wedge y = b) \to \{x := x + y\|\ \{x := x + y\}\ y := x - y\}\ (\{x := x - y\}\ (x = b \wedge y = a)) \\
\rightsquigarrow (x = a \wedge y = b) \to \{x := x + y\|\ y = x + y - y\}\ (\{x := x - y\}\ (x = b \wedge y = a)) \\
\rightsquigarrow (x = a \wedge y = b) \to \{x := x + y\|\ y = x + y - y\}\ (x - y = b \wedge y = a) \\
\rightsquigarrow (x = a \wedge y = b) \to (x + y - (x + y - y) = b \wedge x + y - y = a)
\end{array}$$

which is obviously true.

In Section 6 this system will, together with the update simplification rules, be used as a basis for defining (in WhyML) a VCGen that will produce as proof obligations a set of FOL formulas, that can be checked with any standard proof tool or SMT solver. The calculus, a strategy for update simplification, and the VCGen will all be formalized in Why3, and their properties mechanically proved, in sections 4, 5, and 6.

3. Why3 in a Nutshell

The Why3 platform essentially offers the following: a logic language; a programming language (called WhyML); a proof manager; and a graphical IDE. The logic language extends first-order logic (FOL) with algebraic (inductive) types, polymorphism, and inductive predicates. The programming language is an ML functional language with mutable data structures supporting imperative programming, and integrating specification features (programs may contain contracts and other elements like loop invariants, written in the logic language). Why3 proof obligations are created directly from lemmas and goals written in the logic language, and also from WhyML programs, as *verification conditions* for proving programs correct. The proof manager interacts with a plethora of external proof tools, both automated (including all major SMT solvers) and interactive, and also allows users to manage and store proof sessions. A collection of internal proof transformations are also offered in Why3.

There are several important ways in which the two languages interact: first, the logic language is used to write program specifications. Second, programs can be used to construct proofs of logical results; in particular the contract-based mechanism that is used to generate verification conditions for function calls, can be used to realize induction principles required for non-trivial proofs. Third, *pure* program functions may be defined to exist also in the logic namespace, allowing for the same function to be used both in code and in logic.

As an example, consider the following Why3 fragment (part of a module that first imports relevant list library modules, including the definition of the `sorted` and `permut` predicates).

```
predicate is_a_sorting_algorithm (f: list int -> list int) =
  forall l :list int. permut l (f l) /\ sorted (f l)

let rec function insert (i: int) (l: list int) : list int
  requires { sorted l }
  ensures  { sorted result }
  ensures  { permut result (Cons i l) }
= match l with
  | Nil -> Cons i Nil
  | Cons h t -> if i <= h then Cons i l else Cons h (insert i t)
  end

let rec function iSort (l: list int) : list int
  ensures { sorted result /\ permut result l }
= match l with
  | Nil -> Nil
  | Cons h t -> insert h (iSort t )
  end

goal insertion_sort_correct: is_a_sorting_algorithm iSort
```

The `iSort` WhyML function is the well-known insertion sort algorithm on lists, which in turn uses an auxiliary function for sorted insertion. The behavior of both functions is specified by means of contracts introduced by the `requires` and `ensures` keywords. The verification conditions generated from this fragment are all easily proved, and certify the correctness of this sorting algorithm; no further annotations or inductive proof techniques are required – the VC generation mechanism uses the contracts to formulate appropriate induction hypotheses (both functions exhibit simple structural recursion, but the same principle applies regardless of the number or arguments of the recursive calls). Additionally, both functions are pure and can also be used in the logic language, which allows for the `insertion_sort_correct` goal to be formulated.

In the following sections we make extensive use of pure program functions (also known as `let function`s) in logic. Section 4 formalizes WhileDL in the logic language of Why3, and `let function`s are sometimes used, in which the contract can be seen as a lemma involving the function. `lemma function`s are also used: these are pure functions that do not really exist as programs; they are just used to prove results by induction on various entities. In Sections 5 and 6, respectively, the update simplifier and VCGen are defined as `let function`s, but we have also defined a program-only version of the VCGen, involving mutable types, that can be extracted to OCaml code.

The formalization described in the next sections was developed using Why3 release 1.4.0, and three SMT solvers: Alt-Ergo 2.4.0, CVC4 1.8, and Z3 4.8.6. When we say that a verification condition has been proved automatically, we mean that it has been proved by one of these solvers using Why3's automated strategies. These strategies perform some basic proof transformations (such as splitting) and send simplified goals to the provers with some timeout limits, possibly repeating these steps if any goals remain unproved.

4. Formalization of the Dynamic Logic

In the next sections "..." is often used to indicate parts of the modules that are omited and can be found in the online respository.

theory Exprs_Updates. Our formalization starts with the mutually-recursive definition of program expressions and updates. We also define (mutually-recursive) logic functions `sizeE` and `sizeU` that calculate respectively the size of expressions and updates, and give two recursive `lemma function`s that constitute a (mutually inductive) proof of nonnegativity of the size of expressions and updates. Note how their contracts correspond to the logic results being proved, and the WhyML definitions capture the induction principles being used. Why3 generates verification conditions from these contracts, whose validity ensures the validity of the logic results expressed as postconditions.

```
type ident = | MkIdent int
type operator = Oplus | Ominus | Omult
type expr = | Econst int
            | Evar ident
            | Ebin expr operator expr
            | Eupd upd expr
with upd = | Uskip
           | Uassign ident expr
           | Upar upd upd
           | Uupd upd upd

function sizeE (e:expr) : int =
match e with | Econst _ -> 1 | Evar _ -> 1
             | Ebin e1 _ e2 -> 1 + sizeE e1 + sizeE e2 | Eupd u e -> sizeU u + sizeE e
end with sizeU (u:upd) : int =
match u with | Uskip -> 1 | Uassign _ e -> 1 + sizeE e
             | Upar u1 u2 -> 1 + sizeU u1 + sizeU u2 | Uupd u u' -> 1 + sizeU u + sizeU u'
end

let rec lemma sizeU_pos (u:upd)
  ensures { sizeU u >= 0 }
= match u with | Uskip -> ()   | Uassign _ e -> sizeE_pos e
               | Upar u1 u2 -> sizeU_pos u1 ; sizeU_pos u2
               | Uupd u u' -> sizeU_pos u ; sizeU_pos u'
  end
with lemma sizeE_pos (e:expr)
  ensures { sizeE e >= 0 }
= match e with | Econst _ -> ()   | Evar _ -> ()
               | Ebin e1 _ e2 -> sizeE_pos e1 ; sizeE_pos e2
               | Eupd u e -> sizeU_pos u ; sizeE_pos e
  end
```

We also define (a straightforward recursive definition, not shown) a predicate `indom` expressing when an identifier is in the domain of an update. Our next step is to define evaluation of expressions and updates in a state (defined as a mapping from identifiers to integers), and to write a `lemma function` describing the effect of an update on variables in (resp. not in) its domain. We also define Boolean expressions and their evaluation.

```
type state = map ident int

function eval_bin (x:int) (op:operator) (y:int) : int =
match op with | Oplus -> x+y | Ominus -> x-y | Omult -> x*y end

function eval_expr (s:state) (e:expr) : int =
match e with | Econst n -> n   | Evar x -> get s x
             | Ebin e1 op e2 -> eval_bin (eval_expr s e1) op (eval_expr s e2)
             | Eupd u e -> eval_expr (eval_upd s u s) e
end with eval_upd (s:state) (u:upd) : state -> state =
match u with | Uskip -> fun s' -> s'   | Uassign x e -> fun s' -> set s' x (eval_expr s e)
             | Upar u1 u2 -> fun s' -> eval_upd s u2 (eval_upd s u1 s')
```

```
            | Uupd u1 u2 -> fun s' -> let si = eval_upd s u1 s in eval_upd si u2 s'
end

let rec lemma eval_upd_dom (u:upd)
  ensures { forall s s':state, x:ident. not (indom x u) -> eval_upd s u s' x = s' x }
  ensures { forall s s' s'':state, x:ident. indom x u ->
                                        eval_upd s u s' x = eval_upd s u s'' x }
= match u with | Uskip -> ()   | Uassign _ _ -> ()
               | Upar u1 u2 -> eval_upd_dom u1 ; eval_upd_dom u2
               | Uupd u u' -> eval_upd_dom u ; eval_upd_dom u'
end

type boperator = BOeq | BOlt | BOlteq | BOgt | BOgteq
type bexpr = | Bcomp expr boperator expr | Btrue | Bfalse | Band bexpr bexpr| Bor bexpr bexpr
             | Bnot bexpr | Bupd upd bexpr

predicate eval_bexpr (s:state) (b:bexpr) = ... (* similar to eval_expr *)
end
```

Finally the equivalence predicates `equivUpd`, `equivExp`, and `equivBexp` are defined, to express that the evaluation of two updates or expressions is the same in all states.

theory Programs. The types of program statements and formulas must be defined in a mutually recursive way, since formulas occur in programs (as loop invariant annotations) and programs appear in box modality formulas. For simplicity we consider only quantifier-free first-order formulas. Boolean expressions may be injected into formulas via the `Fembed` constructor. `size` is a `let function` that calculates the size of a program; its definition is close to a count of the number of nodes in the abstract syntax tree, except that in the case of sequences the size of the first statement is given a higher weight. This will be crucial to prove termination of functions whose recursive calls do not always decrease the number of nodes, in particular when a sequence `Sseq (Sseq c1 c2) c` is regrouped as `(Sseq c1 (Sseq c2 c))`. Note also that instead of writing a lemma as done before for `sizeE` and `sizeU`, the function is equipped with a postcondition stating that its result is non-negative.

```
type fmla = | Fembed bexpr | Fsqb stmt fmla | Fupd upd fmla
            | Fand fmla fmla | For fmla fmla | Fnot fmla | Fimplies fmla fmla
with stmt = | Sskip | Sassign ident expr | Sif bexpr stmt stmt
            | Swhile bexpr fmla stmt | Sseq stmt stmt

let rec function size (c:stmt) : int
    ensures { result >= 0 }
= match c with | Sskip -> 1 | Sassign _ _ -> 1 | Sif _ c1 c2 -> 1 + size c1 + size c2
               | Sseq c1 c2 -> 1 + 2*size c1 + size c2 | Swhile _ _ c -> 1 + size c
  end

predicate progInv (c:stmt) =
match c with | Sskip -> True | Sassign _ e -> upd_freeE e
             | Sif b c1 c2 -> upd_freeB b /\ progInv c1 /\ progInv c2
             | Swhile b inv c  -> upd_freeB b /\ stmt_freeF inv /\ upd_freeF inv /\ progInv c
             | Sseq c1 c2 -> progInv c1 /\ progInv c2
end
```

The `progInv` predicate defines *well-formed programs* as those in which expressions do not contain updates, and loop invariants do not contain updates nor box modalities (and thus statements). The predicates `upd_freeE`, `upd_freeB`, `stmt_freeF`, `upd_freeF` are all defined structurally in the obvious way.

The big-step (natural) semantics of programs will be defined by means of an *inductive predicate*, with each clause corresponding to an inference rule. The definition captures the rules of Figure 1.

```
inductive big_step state stmt state =
| big_step_skip: forall s:state. big_step s Sskip s
| big_step_assign: forall s:state, e:expr, x:ident.
        big_step s (Sassign x e) (set s x (eval_expr s e))
| big_step_seq: forall s1 s2 s3:state, c1 c2:stmt.
        big_step s1 c1 s2  -> big_step s2 c2 s3  -> big_step s1 (Sseq c1 c2) s3
| big_step_if_true: forall s s':state, b:bexpr, c1 c2:stmt.
        eval_bexpr s b -> big_step s c1 s' -> big_step s (Sif b c1 c2) s'
| big_step_if_false: forall s s':state, b:bexpr, c1 c2:stmt.
        not (eval_bexpr s b) -> big_step s c2 s' -> big_step s (Sif b c1 c2) s'
| big_step_while_true: forall s s' s'':state, b:bexpr, i:fmla, c:stmt.
        eval_bexpr s b -> big_step s c s'  -> big_step s' (Swhile b i c) s'' ->
        big_step s (Swhile b i c) s''
| big_step_while_false: forall s:state, b:bexpr, i:fmla, c:stmt.
        not (eval_bexpr s b) -> big_step s (Swhile b i c) s
```

The following lemmas about the natural semantics are fairly obvious and proved automatically. They will be useful for proving subsequent results.

```
lemma IfSeqTrue: forall b :bexpr, c1 c2 c :stmt, s s' :state.
  big_step s (Sseq (Sif b c1 c2) c) s' -> eval_bexpr s b -> big_step s (Sseq c1 c) s'

lemma IfSeqFalse: forall b :bexpr, c1 c2 c :stmt, s s' :state.
  big_step s (Sseq (Sif b c1 c2) c) s' -> eval_bexpr s (Bnot b) -> big_step s (Sseq c2 c) s'
```

theory SemanticsL. We define the interpretation of formulas following Section 2 by means of predicate `satisfies` defined below; note that formulas may contain both programs and updates, which must be evaluated. We also define the notions of valid formula (and write a deduction lemma), equivalent formulas, and validity of an update triple $\phi \Longrightarrow \{\mathcal{U}\}\,[C]\,\psi$. Finally, we formulate a lemma relating this interpretation of update triples with the usual interpretation of Hoare triples. Both lemmas are proved automatically.

```
predicate satisfies (s:state) (p:fmla) =
match p with | Fembed b ->  (eval_bexpr s b)
             | Fand p1 p2 -> (satisfies s p1) /\ (satisfies s p2)
         ... | Fsqb c p ->  forall s' :state. big_step s c s' -> satisfies s' p
             | Fupd u p ->  satisfies (eval_upd s u s) p
end

predicate valid_fmla (p:fmla) = forall s:state. satisfies s p
predicate equiv (p:fmla) (q:fmla) = forall s :state. satisfies s p <-> satisfies s q
lemma deduction: forall p q :fmla. (forall s: state. satisfies s p -> satisfies s q)
                                   <-> valid_fmla (Fimplies p q)

predicate validUT (p:fmla)(u:upd)(c:stmt)(q:fmla) =
  valid_fmla (Fimplies p (Fupd u (Fsqb c q)))

lemma validUT_triple : forall p:fmla, u:upd, c:stmt, q:fmla.
  (validUT p  u c q)
    <->
  (forall s s':state. satisfies s p -> big_step (eval_upd s u s) c s' -> satisfies s' q)
```

theory SystemDL. We now need to formulate a fundamental lemma, corresponding to a semantic formulation of the classic Hoare logic rule for while

loops. Its proof is the only one in this first part of the formalization that requires the use of a Why3 proof transformation, specifically for *predicate induction*, available in Why3 through the `induction_pr` transformation:

```
lemma core_while_rule: forall c:stmt, b:bexpr, inv ainv :fmla.
       (forall s s':state. satisfies s (Fand inv (Fembed b)) ->
          big_step s c s'->  satisfies s' inv) ->
        forall s s':state. satisfies s inv ->
          big_step s (Swhile b ainvc) s' -> satisfies s' (Fand inv (Fnot (Fembed b)))
```

We could now formulate as lemmas the inference rules of the WhileDL symbolic execution calculus (Figure 3), and prove them automatically. For instance:

```
lemma seq_while_rule: forall p q inv ainv:fmla, c cc:stmt, b:bexpr, u:upd.
      valid_fmla (Fimplies p (Fupd u inv)) ->
      validUT (Fand inv (Fembed b)) Uskip c inv ->
      validUT (Fand inv (Fnot (Fembed b))) Uskip cc q ->
      validUT p u (Sseq (Swhile b ainv c) cc) q
```

We will however take a different approach and define the calculus using an *inductive predicate* as follows.

```
inductive infUT fmla upd stmt fmla =
| infUT_skip: forall p q:fmla, u:upd.
        valid_fmla (Fimplies p (Fupd u q)) -> infUT p u Sskip q
| infUT_assign: forall p:fmla, q:fmla, x:ident, e:expr, u:upd.
        valid_fmla (Fimplies p (Fupd u (Fupd (Uassign x e) q))) -> infUT p u (Sassign x e) q
| infUT_if: forall p q:fmla, c1 c2 :stmt, b:bexpr, u:upd.
        infUT (Fand p (Fupd u (Fembed b))) u c1 q ->
        infUT (Fand p (Fupd u (Fnot (Fembed b)))) u c2 q ->
        infUT p u (Sif b c1 c2) q
| infUT_while: forall p q:fmla, c :stmt, b:bexpr, inv ainv:fmla, u:upd.
        valid_fmla (Fimplies p (Fupd u inv)) ->
        infUT (Fand inv (Fembed b)) Uskip c inv ->
        valid_fmla (Fimplies (Fand inv (Fnot (Fembed b))) q) ->
        infUT p u (Swhile b ainv c) q
| infUT_skipseq: forall p q:fmla, u:upd, c :stmt.
        infUT p u c q -> infUT p u (Sseq Sskip c) q
| infUT_assignseq: forall p:fmla, q:fmla, x:ident, e:expr, c:stmt, u:upd.
        infUT p (Upar u (Uupd u (Uassign x e))) c q -> infUT p u (Sseq (Sassign x e) c) q
| infUT_ifseq: forall p q:fmla, c1 c2 c:stmt, b:bexpr, u:upd.
        infUT (Fand p (Fupd u (Fembed b))) u (Sseq c1 c) q ->
        infUT (Fand p (Fupd u (Fnot (Fembed b)))) u (Sseq c2 c) q ->
        infUT p u (Sseq (Sif b c1 c2) c) q
| infUT_whileseq: forall p q:fmla, c cc:stmt, b:bexpr, inv ainv :fmla, u:upd.
        valid_fmla (Fimplies p (Fupd u inv)) -> infUT (Fand inv (Fembed b)) Uskip c inv ->
        infUT (Fand inv (Fnot (Fembed b))) Uskip cc q ->
        infUT p u (Sseq (Swhile b ainv c) cc) q
| infUT_seqseq: forall p q:fmla, c1 c2 c:stmt, u:upd.
        infUT p u (Sseq c1 (Sseq c2 c)) q -> infUT p u (Sseq (Sseq c1 c2) c) q
```

Formulas that are checked for validity using the above system are general WhileDL formulas. Later when we formulate a VCGen we will see that under some restrictions on the conclusion update triple, those formulas will be pure FOL formulas (i.e. formulas not containing updates or statements).

theory ReverseRules. Before we can formulate and prove a soundness and completeness result we need a few additional lemmas. First we note that, contrary to Hoare logic, expressiveness in dynamic logic is "built in", simply because the `Fsqb` operator allows for weakest preconditions to be expressed. We express this as a lemma using a weakest precondition state predicate `pre`. Lemmas

`while_rule_revFsqb` and `seq_while_rule_rev_Fsqb`, on the other hand, are "reverse rules" of WhileDL, required to prove completeness.

```
predicate pre (s:state) (c:stmt) (q:fmla) =
         forall s' :state. big_step s c s' -> satisfies s' q

lemma expressiveness : forall c :stmt, q :fmla. forall s :state.
                       (satisfies s (Fsqb c q)) <-> pre s c q

lemma while_rule_revFsqb: forall c:stmt, u:upd, b:bexpr, ainv p q :fmla.
      validUT p u (Swhile b ainv c) q ->
      let inv = Fsqb (Swhile b ainv c) q in
      valid_fmla (Fimplies p (Fupd u inv)) /\
      validUT (Fand inv (Fembed b)) Uskip c inv /\
      valid_fmla (Fimplies (Fand inv (Fnot (Fembed b))) q)

lemma seq_while_rule_rev_Fsqb: forall p q ainv:fmla, c cc:stmt, b:bexpr, u:upd.
      validUT p u (Sseq (Swhile b ainv c) cc) q ->
      let inv = Fsqb (Sseq (Swhile b ainv c) cc) q in
      valid_fmla (Fimplies p (Fupd u inv)) /\
      validUT (Fand inv (Fembed b)) Uskip c inv /\
      validUT (Fand inv (Fnot (Fembed b))) Uskip cc q
```

theory DLSoundnessCompleteness. Soundness and completeness (the specific relative notion known as "box-completeness") of the WhileDL calculus can be expressed as an equivalence result between validity and derivability of an update triple. We will write a `lemma function` having this equivalence as postcondition. The definition of the function will follow the "symbolic execution" structure of the rules, regrouping statements in the case of the sequence constructor. Termination is proved by providing a variant using the `size let function`, which assigns a higher weight to the first statement in the sequence. The VCs generated by Why3 are all proved automatically after some splitting, using one of the tool's automated strategies.

```
let rec lemma infUT_sound_complete (c:stmt) =
    ensures { forall p q :fmla, u :upd. validUT p u c q <-> infUT p u c q  }
    variant { size c }
match c with
| Sskip -> ()
| Sassign _ _ -> ()
| Sif _ c1 c2 -> infUT_sound_complete c1 ; infUT_sound_complete c2
| Swhile _ _ c -> infUT_sound_complete c
| Sseq Sskip c -> infUT_sound_complete c
| Sseq (Sassign _ _) c -> infUT_sound_complete c
| Sseq (Sif _ c1 c2) c -> infUT_sound_complete (Sseq c1 c) ; infUT_sound_complete (Sseq c2 c)
| Sseq (Swhile _ _ c1) c -> infUT_sound_complete c1 ; infUT_sound_complete c
| Sseq (Sseq c1 c2) c -> infUT_sound_complete (Sseq c1 (Sseq c2 c))
end
```

5. Update Simplification

In this section we start constructing a VCGen for WhileDL update triples: an algorithm that generates a set of first-order proof obligations, sufficient to ensure the validity of the triple. For this we will apply the rules of the calculus of Figure 3; note however that this is not sufficient, since rules (skip), (assign), and (while) have premises that are general formulas of WhileDL, possibly containing

updates. So the first step is to write a formula simplifier, a function with the following contract:

```
val function simplF (p: fmla) : fmla =
    requires { stmt_freeF p }
    ensures  { stmt_freeF result /\ upd_freeF result /\ equiv p result }
```

Note that the simplifier only works with formulas not containing box modalities (which is the case in pre- and post-conditions in WhileDL update triples). It will produce an equivalent formula that is free of updates.

The simplification procedure is based on a strategy for the rules of Figure 2: updates are first simplified to equivalent parallel updates that are propagated through the structure of the formulas; when a parallel update is applied to a variable, a substitution is performed by looking up the variable in the update.

module UpdateApplication. We first define the notion of normalized parallel update (a right spline of `Uassign` constructors with no occurrences of updates in expressions, with the same left-hand side possibly occuring more than once), a function for composing parallel updates, and another for looking up a variable in a parallel update (with a "righmtost wins" semantics). Since they will be used by the VCGen to be defined later, the latter two are defined as program functions, and equipped with appropriate contracts; the VCs pertaining to these contracts are automatically proved.

```
predicate parUpd (u:upd) =
match u with | Uskip -> true | Uassign _ e -> upd_freeE e
             | Upar (Uassign _ e) u -> upd_freeE e /\ parUpd u | _ -> False
end

let rec function concat (u1:upd) (u2:upd) : upd
    requires { parUpd u1 /\ parUpd u2 }
    ensures  { parUpd result /\ equivUpd result (Upar u1 u2) }
    variant  { u1, u2 }
= match u1 with | Uskip -> u | Uassign _ _ -> Upar u1 u2
                | Upar u1a u1b -> concat u1a (concat u1b u2) | _ -> absurd
  end

let rec lookup (u:upd) (x:ident) : expr
    requires { parUpd u }
    ensures  { equivExp result (Eupd u (Evar x)) /\ upd_freeE result }
    variant  { u }
= match u with | Uskip -> Evar x | Uassign y e' -> if eq x y then e' else Evar x
               | Upar (Uassign y e) ub -> if eq x y && not (indom_exec x ub) then e
                                          else lookup ub x
               | _ -> absurd
  end
```

The next set of functions apply a parallel update to expressions, updates (of any kind), and formulas, resulting in entities that are free from updates and equivalent to an update constructor application.

```
let rec function applyE (u:upd) (e:expr) : expr
    requires { parUpd u }
    ensures  { equivExp result (Eupd u e) /\ upd_freeE result }
    variant  { sizeE e }
= match e with | Econst n -> Econst n | Evar x -> lookup u x
               | Ebin e1 op e2 -> Ebin (applyE u e1) op (applyE u e2)
               | Eupd u' e' -> applyE (concat u (applyU u u')) e'
  end
```

```
with function applyU (u:upd) (u':upd) : upd
    requires { parUpd u }
    ensures  { parUpd result /\ equivUpd result (Uupd u u') }
    variant  { sizeU u' }
= match u' with | Uskip -> Uskip | Uassign x e -> Uassign x (applyE u e)
                | Upar ua ub -> concat (applyU u ua) (applyU u ub)
                | Uupd ua ub -> applyU (concat u (applyU u ua)) ub
  end

let rec function  applyB (u: upd) (b: bexpr) : bexpr
  (...)

let rec function applyF (u: upd) (p: fmla) : fmla
    requires { parUpd u /\ stmt_freeF p }
    ensures  { stmt_freeF result /\ upd_freeF result /\ equiv result (Fupd u p) }
    variant  { p }
= match p with | Fcomp e1 bop e2 -> Fcomp (applyE u e1) bop (applyE u e2)
               | Fembed b ->  Fembed (applyB u b)
               | Fand p1 p2 ->  Fand (applyF u p1) (applyF u p2)
           ... | Fsqb _ _ -> absurd
               | Fupd u' p -> applyF (concat u (applyU u u')) p
  end
```

module Simplification. We may now define the simplification function for formulas, together with auxiliary simplification functions for expressions and updates. Simplified expressions and formulas do not contain updates, and the simplification of updates produces parallel updates. In the case of update applications, all three functions first simplify updates to parallel form before invoking an `apply_` function to proceed with the simplification. Integer expressions and updates are, as expected, simplified in a mutually recursive way. All functions are automatically proved to be correct.

```
let rec function simplE (e:expr) : expr
    ensures { upd_freeE result /\ equivExp e result }
    variant { e }
= match e with | Econst n -> Econst n | Evar x -> Evar x
               | Ebin e1 op e2 -> Ebin (simplE e1) op (simplE e2)
               | Eupd u e' -> applyE (simplU u) e'
  end with function simplU (u:upd) : upd
    ensures { parUpd result /\ equivUpd u result }
    variant { u }
= match u with | Uskip -> Uskip
               | Uassign x e -> Uassign x (simplE e)
               | Upar u1 u2 -> let u1s = simplU u1 in let u2s = simplU u2 in concat u1s u2s
               | Uupd u1 u2 -> let u1s = simplU u1 in applyU u1s u2
  end

let rec function  simplB (b: bexpr) : bexpr  (...)

let rec function simplF (p: fmla) : fmla
    requires { stmt_freeF p }
    ensures  { stmt_freeF result /\ upd_freeF result /\ equiv p result }
    variant  { p }
= match p with | Fcomp e1 bop e2 -> Fcomp (simplE e1) bop (simplE e2)
| Fembed b ->  Fembed (simplB b) | Ftrue -> Ftrue | Ffalse -> Ffalse
               | Fand p1 p2 ->  Fand (simplF p1) (simplF p2)
           ... | Fsqb _ _ -> absurd | Fupd u' p -> applyF (simplU u') p
  end
```

6. Verification Conditions

The VCGen is not meant for general update triples, but specifically for update triples $\phi \Longrightarrow \{\mathcal{U}\}\,[C]\,\psi$ satisfying the following restrictions:

- ϕ and ψ do not contain statements;
- ϕ does not contain updates;
- the update $\mathcal{U}$ is in parallel normalized form (i.e. it is a sequence of elementary updates $x_1 := a_1 \| \ldots \| x_n := a_n$);
- C is well-formed in the sense of predicate `progInv`.

The Hoare triple $\{\phi\}\,C\,\{\psi\}$ can be seen as a particular case of the update triple $\phi \Longrightarrow \{\mathcal{U}\}\,[C]\,\psi$ where the update is `skip`, and thus the above restrictions are met (and preserved by recursive calls of the VCGen).

The output of the VCGen will be a set of FOL formulas. We first define a type for these formulas, which are basically WhileDL formulas without occurrences of modalities or updates. We define predicates for satisfaction and validity of FOL formulas, together with a predicate expressing equivalence between FOL and WhileDL formulas, and a conversion function from statement- and update-free WhileDL formulas to FOL formulas.

```
type fmlaFOL = | FOLcomp expr boperator expr | FOLembed bexpr | FOLtrue | FOLfalse
               | FOLand fmlaFOL fmlaFOL | FOLor fmlaFOL fmlaFOL
               | FOLnot fmlaFOL | FOLimplies fmlaFOL fmlaFOL

predicate satisfiesFOL (s:state) (p:fmlaFOL) =
match p with | FOLembed b ->  (eval_bexpr s b)
         ... | FOLand p1 p2 -> (satisfiesFOL s p1) /\ (satisfiesFOL s p2)
end

predicate valid_fmlaFOL (p:fmlaFOL) = forall s:state. satisfiesFOL s p

predicate equivFOL (p:fmlaFOL) (q:fmla) = forall s :state. satisfiesFOL s p <-> satisfies s q

let rec function toFOL (p:fmla) : fmlaFOL
    requires { stmt_freeF p /\ upd_freeF p }
    ensures  { equivFOL result p }
    variant  { p }
= match p with | Fembed b ->  FOLembed b | Fand p1 p2 -> FOLand (toFOL p1) (toFOL p2)
           ... | Fsqb _ _ ->  absurd | Fupd _ _ ->  absurd
  end
```

We use the `fset` polymorphic type for finite sets from the Why3 library, which makes available in particular the functions `singleton`, `union`, and `add` with the obvious meanings.

The `vcgen` function follows the rules of Figure 3, collecting (after simplfication) the formulas in the leaves of the derivation. Whenever an update is applied in a rule, its application is immediately simplified. The function's contract expresses the correctness of the VCGen: if the generated FOL formulas are all valid, then the input WhileDL update triple is indeed valid.

```
predicate valid_fmlas (g: fset fmlaFOL) = forall p :fmlaFOL. mem p g -> valid_fmlaFOL p
```

```
let ghost function singletonFOL (p:fmla) : fset fmlaFOL = singleton (toFOL p)

let ghost function addFOL (p:fmla) (v:fset fmlaFOL) : fset fmlaFOL = add (toFOL p) v

let rec ghost function vcgen (p:fmla) (u:upd) (c:stmt) (q:fmla) : fset fmlaFOL
    requires { stmt_freeF p /\ upd_freeF p /\ parUpd u /\ progInv c /\ stmt_freeF q }
    ensures  { valid_fmlas result -> validUT p u c q }
    variant  { size c }
= match c with
  | Sskip -> singletonFOL (Fimplies p (applyF u q))
  | Sassign x e -> singletonFOL (Fimplies p (applyF u (applyF (Uassign x e) q)))
  | Sif b c1 c2 -> union (vcgen (Fand p (applyF u (Fembed b))) u c1 q)
                    (vcgen (Fand p (applyF u (Fnot (Fembed b)))) u c2 q)
  | Swhile b inv c1 -> addFOL (Fimplies p (applyF u inv))
                        (addFOL (Fimplies (Fand inv (Fnot (Fembed b))) (simplF q))
                           (vcgen (Fand inv (Fembed b)) Uskip c1 inv))
  | (Sseq (Sskip) c) -> vcgen p u c q
  | (Sseq (Sassign x e) c) -> vcgen p (concat u (applyU u (Uassign x e))) c q
  | (Sseq (Sif b c1 c2) c) ->  union (vcgen (Fand p (applyF u (Fembed b))) u (Sseq c1 c) q)
                                (vcgen (Fand p (applyF u (Fnot (Fembed b)))) u (Sseq c2 c) q)
  | (Sseq (Swhile b inv c1) c) -> addFOL (Fimplies p (applyF u inv))
                                    (union (vcgen (Fand inv (Fembed b)) Uskip c1 inv)
                                       (vcgen (Fand inv (Fnot (Fembed b))) Uskip c q))
  | (Sseq (Sseq c1 c2) c) -> vcgen p u (Sseq c1 (Sseq c2 c)) q
end
```

7. Conclusion

This paper can be seen from two different perspectives, with different contributions. The first contribution is a non-trivial case study in program verification with Why3. In this light, we verify a functional program, consisting of the VCGen in Section 6 and the simplification functions of Section 5, with a complex specification described in Section 4. In the online repository the reader can find a final step of the development, not included here for lack of space: we write an execution version of the VCGen, replacing the abstract `fset` type of polymorphic sets with the concrete Why3 library type `SetImp` for mutable sets. Because it uses a mutable type, this alternative VCGen can no longer be used as a logic function; it can, however, be extracted to OCaml code using Why3's program extraction facility, resulting in an actual executable, correct-by-construction VCGen.

A second contribution of the paper, at the program logic level, is the design of the VCGen itself, which produces first-order verification conditions for a Hoare triple in a forward way, semantically justified by the underlying dynamic logic. Although the language considered here does not make use of updates for handling reference aliasing, this is still a sufficient proof of concept of how a dynamic logic-based verifier can be constructed making used of standard first-order proof tools. It will be interesting to compare the generated VCs with those that are obtained by other methods (for instance in terms of size), but an immediate observation is that, unlike the traditional forward propagation method based on a strongest postcondition predicate transformer, no existential quantifiers are introduced. On the other hand, no multiple version variables need to be introduced, as in methods based on the use of single-assignment intermediate forms [11].

The Why3 files containing the above modules are available from the repository `https://github.com/jspdium/dlKeY`, which also includes proof session folders and html proof summaries.

References

[1] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, P. H. Schmitt, The KeY tool, Software and System Modeling 4 (1) (2005) 32–54.

[2] C. A. R. Hoare, An axiomatic basis for computer programming, Communications of the ACM 12 (1969) 576–580.

[3] E. W. Dijkstra, A Discipline of Programming, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[4] J. C. Reynolds, Separation logic: A logic for shared mutable data structures, in: LICS, IEEE Computer Society, 2002, pp. 55–74.

[5] D. Harel, First Order Dynamic Logic, Vol. 68 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1979.

[6] D. Harel, D. Kozen, J. Tiuryn, Dynamic logic, in: Handbook of Philosophical Logic, MIT Press, 1984, pp. 497–604.

[7] B. Beckert, A. Platzer, Dynamic logic with non-rigid functions, in: U. Furbach, N. Shankar (Eds.), Automated Reasoning, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 266–280.

[8] B. Beckert, V. Klebanov, B. Weiß, Dynamic Logic for Java, Springer International Publishing, Cham, 2016, pp. 49–106.

[9] J.-C. Filliâtre, A. Paskevich, Why3 — where programs meet provers, in: M. Felleisen, P. Gardner (Eds.), Proceedings of the 22nd European Symposium on Programming, Vol. 7792 of Lecture Notes in Computer Science, Springer, 2013, pp. 125–128.

[10] R. Hähnle, R. Bubel, A Hoare-style calculus with explicit state updates, department of Computer Science, Chalmers University of Technology.

[11] C. B. Lourenço, M. J. Frade, J. S. Pinto, Formalizing single-assignment program verification: an adaptation-complete approach, in: P. Thiemann (Ed.), Proceedings of the 25th European Symposium on Programming (ESOP 2016), Vol. 9632 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, 2016, pp. 41–67.