A Dynamic Logic for Unstructured Programs with Embedded Assertions

Mattias Ulbrich

Karlsruhe Institute of Technology
Institute for Theoretical Computer Science
D-76128 Karlsruhe, Germany
ulbrich@kit.edu

Abstract. We present a program logic for an intermediate verification programming language and provide formal definitions of its syntax and semantics. The language is unstructured, indeterministic, and has embedded assertions. A set of sound rewrite rules which allow symbolic execution of programs is given. We prove the soundness of three inference rules using invariants which can be used to deal with loops during the verification.

1 Introduction

The purpose of deductive software verification is to formally prove that a piece of code in a particular programming language behaves as specified. This can be done on the level of the programming language or after a translation to an intermediate verification language. In this paper, we will consider a minimalistic, general intermediate verification language that covers the essential features of established intermediate languages and is close to Boogie [10]. We present a program logic in the style of first-order dynamic logic (DL).

DL is a program logic which embeds pieces of code within formulas. In its original presentation [9] by Harel et al., a code fragment π in a structural language gives rise to a modality $[\pi]$ which can be used as a prefix to a formula ϕ. The result is the formula $[\pi]\phi$ which is true if and only if ϕ holds in every state in which the execution of π terminates. The Hoare triple $\{\psi\}\pi\{\phi\}$ can hence be written as $\psi \to [\pi]\phi$ in DL. Since every intermediate step of a symbolic execution in DL is a formula itself, this type of verification allows the alternation of symbolic execution and the application of deductive inference rules. Therefore, symbolically stepping through a program provides further insight into a process which usually happens hidden in the verification condition generator. This is not only helpful for finding mismatches between specification and implementation, but also particularily valuable when experimenting with new modelling or translation techniques. Other approaches use weakest precondition (wp) calculi to automatically generate first order verification conditions. In the end, automatic generation and proving of first-order verification conditions as done by these approaches is certainly preferable, but we believe that, in the present state of research, the possibility of interaction is a valuable factor.

An intermediate verification language provides – like the intermediate representation for compilers – a common verification platform for different programming languages and specification techniques. If it is general enough, many programming languages can be translated to it, thereby decoupling the semantics of the source language from the actual verification process.

The design of the present intermediate language requires considerable adaptations of the original DL. Moreover, we give a set of sound rewrite rules which allow symbolic execution of programs, and prove the soundness of three inference rules which can be used to deal with loops using invariants.

We present the dynamic logic in Sect. 2. The rewrite rules used to symbolically execute programs in formulas are given in Sect. 3. Gentzen-style inference rules for the treatment of loops are presented and proved correct in Sect. 4. An overview of related work in Sect. 5 and conclusions in Sect. 6 wrap up the paper.

2 Syntax and Semantics

In this section, we present the syntax and semantics of unstructured dynamic logic (*USDL*). It is built around a minimalistic intermediate verification language which is unstructured, indeterministic and contains embedded assertions. The logic extends untyped first-order predicate logic, but the approach can easily be transferred to sorted logics, the issue of types is orthogonal to the novelties presented here. For instance, the polymorphic type system presented in [11] could be used.

Unlike in DL where a program π can be used as a prefix $[\pi]$ to a formula, in *USDL* π and a natural number n induce an *atomic program formula* $[n;\pi]$ which is not prefix to another formula but a formula on its own. The number n is an explicitly denoted program pointer referring to the currently active statement in π. The conditions that we want to check are embedded within π. This is done because it is not always the case that we only need to examine whether properties hold *after* the execution, but often want to ensure that properties hold at certain points *during* the execution of a program.

For instance, if a program contains a division expression $1/x$, we need to verify that x is different from 0 to ensure its correctness. This check cannot be postponed to the after state of the entire code, but needs to performed in the state in which the expression is evaluated. This can be addressed by inserting an assertion at the appropriate place into the intermediate code.

2.1 Syntax

USDL is an extension of first order logic with two additional modal operators. Besides the atomic program formulas, we introduce the concept of updates which are explicitly denoted value assignments to record the effect of assignment statements.

Definition 1 (Signature). *A USDL-signature* $\Sigma = (\mathrm{Var}, \mathrm{Fct}, \mathrm{PVar}, \mathrm{Prd}, \alpha)$ *is a 5-tuple with*

$$
\begin{array}{lll}
\text{Formula} & ::= & \text{Formula}\ \big(\wedge \mid \vee \mid \rightarrow\big)\ \text{Formula} \\
 & \mid & \neg\ \text{Formula} \mid \textbf{true} \mid \textbf{false} \\
 & \mid & (\forall \mid \exists)\ \mathit{Var}\ \textbf{.}\ \text{Formula} \\
 & \mid & \mathit{Prd} \quad\mid\quad \mathit{Prd}\ \textbf{(}\ \text{TermList}\ \textbf{)} \qquad (*) \\
 & \mid & \textbf{\{}\ \text{Update}\ \textbf{\}}\ \text{Formula} \\
 & \mid & [\ \mathit{NaturalNumber}\ \textbf{;}\ \text{Program}\ ] \\
 & \mid & [[\ \mathit{NaturalNumber}\ \textbf{;}\ \text{Program}\ ]] \\
\text{Term} & ::= & \mathit{Var} \quad\mid\quad \mathit{Fct} \quad\mid\quad \mathit{Fct}\ \textbf{(}\ \text{TermList}\ \textbf{)} \qquad (*) \\
 & \mid & \textbf{\{}\ \text{Update}\ \textbf{\}}\ \text{Term} \\
\text{TermList} & ::= & \text{Term} \mid \text{TermList}\ \textbf{,}\ \text{TermList} \\
\text{Update} & ::= & \mathit{PVar}\ \textbf{:=}\ \text{Term} \quad\mid\quad \text{Update} \parallel \text{Update} \\
\text{Program} & ::= & \text{Statement} \quad\mid\quad \text{Program}\ \textbf{,}\ \text{Program} \\
\text{Statement} & ::= & \mathit{PVar}\ \textbf{:=}\ \text{Term} \\
 & \mid & \textbf{assert}\ \text{Formula} \qquad\qquad \mid \textbf{assume}\ \text{Formula} \\
 & \mid & \textbf{havoc}\ \mathit{PVar} \qquad\qquad\quad \mid \textbf{goto}\ \mathit{NaturalNumber} \\
 & \mid & \textbf{goto}\ \mathit{NaturalNumber}\ \mathit{NaturalNumber}
\end{array}
$$

(*) if the length of the term list coincides with the arity of the symbol

Fig. 1. Formulas, Terms and Programs

- Var*: the set of logical variable symbols*
- Fct*: the non-empty set of function symbols*
- PVar $\subseteq$ Fct*: the set of program variables*
- Prd*: the set of predicate symbols*
- $\alpha : \mathrm{Fct} \cup \mathrm{Prd} \to \mathbb{N}$*: the arity mapping*
- $\alpha(pv) = 0$ *for any program variable* $pv \in \mathrm{PVar}$

The syntax of terms, formulas and programs is given by the grammar in Fig. 1. For predicate and function application expressions, we additionally insist on a correct number of argument terms. If a predicate or function symbol s has no arguments, we write s instead of $s()$. Terminal symbols are set in *italics* and terminal literals in **bold**.

Definition 2 (Terms and Formulas). *The set* $Term_\Sigma$ *of all terms in the signature* Σ *is the set of expressions which can be produced from the non-terminal "Term" in Fig. 1.*

The set $Form_\Sigma$ *of all formulas in the signature* Σ *is the set of expressions which can be produced from the non-terminal "Formula" in Fig. 1.*

Let us for an example consider a *USDL*-signature Σ which contains a program variable $x \in \mathrm{PVar}$, a unary predicate symbol $pos \in \mathrm{Prd}$ and a unary function symbol $suc \in \mathrm{Fct}$. The expression

$$[0; \mathsf{goto}\ 1\ 4, \mathsf{assume}\ \neg pos(x), x := suc(x), \mathsf{goto}\ 0,\\ \mathsf{assume}\ pos(x), \mathsf{assert}\ pos(x)] \quad (1)$$

is then a valid atomic program formula in Form_Σ. In Ex. 12 we show how this formula can be used for the symbolic execution of the contained program.

Definition 3 (Unstructured programs). *The set of all unstructured programs $\varPi_\Sigma$ is the set of expressions that can be produced from the non-terminal "Program" in Fig. 1. Terms and formulas that are embedded in unstructured programs must not have free variables.*

For a given program $\pi \in \varPi_\Sigma$, $len(\pi) \in \mathbb{N}$ denotes the length (i.e., the number of statements) of π. For a natural number $i \in \mathbb{N}$, the selection $\pi[i]$ refers to the i-th statement in π if $i < len(\pi)$ and refers to the statement "`assume` false*" if $i \geq len(\pi)$.*

Unlike in dynamic logic for structured programs, we need to include statements located before the active statement in the modalities. This is because `goto` statements may refer to any position in the program, before or after the current one. We employ an explicit program counter indicating current statement.

2.2 Semantics

We start the definition of our model-theoretic semantics by repeating the definition of first order structures.

Definition 4 (Domain, Interpretation, Variable assignment). *A* domain *$\mathcal{D}$ is a non-empty set. For a given domain $\mathcal{D}$ and a signature Σ an* interpretation *I is a mapping assigning a meaning to every predicate and function symbol in Σ, such that*

- $I(f) : \mathcal{D}^{\alpha(f)} \to \mathcal{D}$ *for any* $f \in \mathrm{Fct}$
- $I(p) \subseteq \mathcal{D}^{\alpha(p)}$ *for any* $f \in \mathrm{Prd}$

A variable assignment *$\beta : Var \to \mathcal{D}$ is a mapping from the logical variables to elements in the domain.*

The set of all interpretation functions for a given $\mathcal{D}$ and Σ is denoted by $\mathcal{I}_{\Sigma,\mathcal{D}}$.

For the notion of the state of an execution of an unstructured program, we need a way to refer to the current position within the sequence of statements, i.e. a program counter pointing to the active statement.

Definition 5 (State). *For a signature Σ and a domain $\mathcal{D}$, the set of states $\mathcal{S}_{\Sigma,\mathcal{D}} := \mathcal{I}_{\Sigma,\mathcal{D}} \times \mathbb{N}$ is the Cartesian product of interpretations (current variable state) and natural numbers (current position in the program).*

We explicitly encode the current statement number within the execution state as it simplifies the definition of state transitions considerably if the execution environment includes a reference to the statement to be executed next.

Definition 6 (Function overriding). *Given a function $f : A \to B$ and values $a \in A$ and $b \in B$ the* function overriding *$f_a^b : A \to B$ is the function with*

$$f_a^b(x) = \begin{cases} b & \textit{if } x = a \\ f(x) & \textit{otherwise} \end{cases} .$$

An update syntactically describes a change of the evaluation state. Applying an update to an evaluation context overrides the interpretation function. Therefore, in the upcoming definition and for an update $c_1 := t_1 \| \ldots \| c_n := t_n$ and an interpretation I we use the notation

$$I^{c_1:=t_1\|\ldots\|c_n:=t_n} := ((I_{c_1}^{val_{I,\beta}(t_1)})\ldots)_{c_n}^{val_{I,\beta}(t_n)}$$

to denote the interpretation in which the symbols $c_1, \ldots, c_n$ have their values updated.

Definition 7 (Term evaluation). *For a given signature Σ, a domain $\mathcal{D}$, an interpretation I and a variable assigment β, the term valuation* $\mathrm{val}_{I,\beta} : Term_\Sigma \to \mathcal{D}$ *is defined by:*

- $\mathrm{val}_{I,\beta}(x) = \beta(x)$ *if* $x \in \mathrm{Var}$,
- $\mathrm{val}_{I,\beta}(f(t_1,\ldots,t_k)) = I(f)(\mathrm{val}_{I,\beta}(t_1),\ldots,\mathrm{val}_{I,\beta}(t_k))$ *if* $f \in \mathrm{Fct}$ *with* $\alpha(f) = k$ *and* $t_1,\ldots,t_k \in Term_\Sigma$,
- $\mathrm{val}_{I,\beta}(\{\mathcal{U}\}t) = val_{I^{\mathcal{U}},\beta}(t)$

For the definition of the semantics of atomic program formulas, the semantics of programs has to be defined. The next two definitions for programs and formulas (Def. 8 and 9) depend on each other and have to be read as one. It may appear counter-intuitive that in Def. 8, the semantics of assert and assume statements seem identical. The difference is that a trace is considered *successful* if it fails at an assumption but *unsuccessful* for a failed assertion.

Definition 8 (Program execution, Traces). *The program execution function $R_\pi : \mathcal{S}_{\Sigma,\mathcal{D}} \to \mathbb{P}(\mathcal{S}_{\Sigma,\mathcal{D}})$ is a mapping that for a program $\pi \in \Pi_\Sigma$ assigns to every state a set of successor states. Its result depends on the currently active statement.*

Let $s = (I, n) \in \mathcal{S}_{\Sigma,\mathcal{D}}$ be a state and β a variable assignment. Then the value of $R_\pi(s)$ is according to the following table:

If $\pi[n]$ ***matches***	***and***	***then*** $R_\pi(s) =$
$c := t$		$\{(I_c^{\mathrm{val}_{I,\beta}(t)}, n+1)\}$
assert ϕ	$I,\beta \models \phi$	$\{(I, n+1)\}$
assert ϕ	$I,\beta \not\models \phi$	$\emptyset$
assume ϕ	$I,\beta \models \phi$	$\{(I, n+1)\}$
assume ϕ	$I,\beta \not\models \phi$	$\emptyset$
goto m		$\{(I, m)\}$
goto m k		$\{(I, m), (I, k)\}$
havoc c		$\{d \in \mathcal{D} \bullet (I_c^d, n+1)\}$

- *We call a sequence $(s_0, s_1, \ldots, s_r)$ (or $(s_0, s_1, \ldots)$ resp.) with $s_i \in \mathcal{S}$ and $s_{i+1} \in R_\pi(s_i)$ for $i \in \{0,...,r-1\}$ (resp. $i \in \mathbb{N}$) a finite (infinite)* trace *of π starting in s_0.*
- *We call a finite trace* maximal *if $R_\pi(s_r) = \emptyset$.*

– *A maximal finite trace* $(s_0, s_1, \dots, s_r)$ *with* $s_r = (I_r, n_r)$ *is called* successful *if* $\pi[n_r]$ *is not an "*assert ...*" statement.*

Unstructured programs are indeterministic, hence, there may be no, one or many successor states in $R_\pi(s)$ to a state s. Two types of indeterminism can be distinguished: control indeterminism (induced by goto statements with two targets) and data indeterminism (induced by havoc statements which take many possible assignments into account).

Definition 9 (Formula evaluation). *For given Σ, I, β, π and $\mathcal{D}$, the validity of a formula $\phi \in Form_\Sigma$ under the given parameters is defined as:*

– $I, \beta \models \mathrm{true}$ *and* $I, \beta \not\models \mathrm{false}$
– $I, \beta \models \phi\,(\wedge|\vee|\to)\,\psi$ *iff* $I, \beta \models \phi$ *and/or/implies* $I, \beta \models \psi$*.*
– $I, \beta \models (\forall|\exists)x.\phi$ *iff* $I, \beta^d_x \models \phi$ *for every/some* $d \in \mathcal{D}$*.*
– $I, \beta \models p(t_1, \dots, t_k)$ *iff* $(\mathrm{val}_{I,\beta}(t_1), \dots, \mathrm{val}_{I,\beta}(t_k)) \in I(p)$ *for a predicate symbol* $p \in \mathrm{Prd}$ *with* $\alpha(p) = k$ *and* $t_1, \dots, t_k \in Term_\Sigma$*.*
– $I, \beta \models \{\mathcal{U}\}\phi$ *iff* $I^{\mathcal{U}}, \beta \models \phi$
– $I, \beta \models [n; \pi]$ *iff every maximal finite trace* $(I, n), \dots, (I_k, n_k)$ *is successful.*
– $I, \beta \models [[n; \pi]]$ *iff* $I, \beta \models [n; \pi]$ *and there is no infinite trace of* π *starting in* (I, n)*.*

Let us revisit example (1) considering an interpretation with the domain $\mathcal{D} = \mathbb{Z}$, $I(succ)(n) = n + 1$ and $I(pos) = \mathbb{N}$. If $I(x) = -1$, we have the maximal trace $(I, 0), (I, 4)$ which is successful since the last considered statement $\pi[4]$ was not an assertion but an assumption. We are not interested in a further execution of this trace and regard it as "not relevant" since an assumption has proved to be false.

USDL possesses expressive means to model both partial and total correctness of code pieces using the operators $[\cdot]$ and $[[\cdot]]$. Please note that they are not dual to another like $\Box$ and $\Diamond$ in modal logics or $[\cdot]$ and $\langle\cdot\rangle$ in classical dynamic logic are.

The programming language of *USDL* has a number of points in common with regular programs upon which the while-language in dynamic logic has been defined in [9]. The program operators $\cup$ (nondeterministic choice) and * (nondeterministic repetition) are closely related to the indeterministic goto statement. The statement assume ϕ has the same semantics as the regular program $\phi?$. Harel et al. also propose an extension with wildcard assignments like $x :=?$ which is the same as the statement havoc x.

Hence, we can use the kinds of statement defined in this document to define compound structures as macros like Harel did using regular programs. Formula (1) could then be reformulated as

$$[0; \mathsf{while}\ \neg pos(x)\ \mathsf{do}\ x := suc(x)\ \mathsf{end}; \mathsf{assert}\ pos(x)] \qquad (2)$$

using such a macro for the while-do-end loop. This formula embeds in a formula the meaning of the Hoare triple $\{\mathrm{true}\}\mathsf{while}\ \neg pos(x)\ \mathsf{do}\ x := suc(x)\ \mathsf{end}\{pos(x)\}$.

3 Symbolic Execution

We now present a set of rewriting rules which allow us to symbolically execute an unstructured program step by step, either interactively or in an automatic proof process. Unlike wp-calculi which traverse programs from back to front, we process programs in the order of an execution, beginning at the first statement. The update mechanism allows us to record the state changes we collect during the execution. This forward treatment is particularly helpful if the execution is part of an interactive verification process since the verifier can then track more conveniently what has happened.

A rewrite rule $l \rightsquigarrow r$ allows the calculus to replace any occurrence of l within a formula with r to obtain an equivalent formula. Such a rule is sound if the formula $l \leftrightarrow r$ is valid. A rule schema of the form $C(X) \implies l(X) \rightsquigarrow r(X)$ with a set of schematic variables X is an abbreviation for the set $\{l(x) \rightsquigarrow r(x) \mid C(x)\}$ of all instances for which the (meta) condition C holds.

Theorem 10 (Symbolic execution). *The following rules are sound rewrite rules for the symbolic execution of unstructured programs.*

$$\pi[n] = c := v \implies [n;\pi] \rightsquigarrow \{c := v\}[n+1;\pi] \tag{3}$$
$$\pi[n] = \mathsf{havoc}\ c \implies [n;\pi] \rightsquigarrow \forall x.\{c := x\}[n+1;\pi] \tag{4}$$
$$\pi[n] = \mathsf{goto}\ m \implies [n;\pi] \rightsquigarrow [m;\pi] \tag{5}$$
$$\pi[n] = \mathsf{goto}\ m\ k \implies [n;\pi] \rightsquigarrow [m,\pi] \wedge [k;\pi] \tag{6}$$
$$\pi[n] = \mathsf{assume}\ \phi \implies [n;\pi] \rightsquigarrow \phi \to [n+1;\pi] \tag{7}$$
$$\pi[n] = \mathsf{assert}\ \phi \implies [n;\pi] \rightsquigarrow \phi \wedge [n+1;\pi] \tag{8}$$

Proof. The soundness proofs for these rules are straightforward. We exemplarily provide them for (7) and (8). The basic argument is the same for all cases: We reduce the case that all finite traces starting in (I, n) must be successful to the case that all finite traces from $(I', n') \in R_\pi(I, n)$ are successful and encode the knowledge on I' either into an update, an implication or conjunction. The state successor relation R_π of assert and assume are identical, but their semantics differ due to the definition of successful traces.

assume: If $I, \beta \not\models \phi$, then $R_\pi(I, n) = \emptyset$ and the only trace beginning in (I, n) ends in an assume statement and, hence, *is* successful. If $I, \beta \models \phi$, the truth value depends entirely on the traces starting in $(I, n + 1)$, therefore, on $[n + 1; \pi]$.

$$\begin{aligned}
& I, \beta \models [n;\pi] \\
\iff & \text{every finite trace beginning in } (I, n) \text{ is successful} \\
\iff & I, \beta \not\models \phi \text{ or} \\
& I, \beta \models \phi \text{ and every finite trace beginning in } (I, n+1) \text{ is successful} \\
\iff & I, \beta \not\models \phi \text{ or every finite trace beginning in } (I, n+1) \text{ is successful} \\
\iff & I, \beta \models \phi \to [n+1;\pi]
\end{aligned}$$

assert: If $I, \beta \not\models \phi$, the only trace beginning in (I, n) ends in an assert statement and, hence, *is not* successful. The other case depends again on the traces from $(I, n+1)$:

$$\begin{aligned}
& I, \beta \models [n; \pi] \\
\iff & \text{every finite trace beginning in } (I, n) \text{ is successful} \\
\iff & I, \beta \models \phi \text{ and every finite trace beginning in } (I, n+1) \text{ is successful} \\
\iff & I, \beta \models \phi \wedge [n+1; \pi]
\end{aligned}$$

□

The presented rules execute one single step and reduce a formula to one encoding *all* possible follow-up traces. This implies that the traces of the atomic program formulas on the left-hand-side are finite if and only if all traces of all modalities on the right-hand-side are finite. This observation leads to

Corollary 11. *We obtain sound rewrite rules if we replace every occurrence of a modality* $[n; \pi]$ *by the corresponding terminating counterpart* $[[n; \pi]]$ *in* (3)–(8).

Example 12. Let us now reconsider formula (1) which is $[0, \pi]$ for the program

$$\pi = \big(\mathsf{goto}\ 1\ 4; \mathsf{assume}\ \neg pos(x); x := suc(x); \mathsf{goto}\ 0; \mathsf{assume}\ pos(x); \mathsf{assert}\ pos(x)\big)\ .$$

By repeatedly applying the calculus rules (3)–(8), we can execute the program, statement by statement resulting in the following chain of equivalent formulas.

$$\begin{aligned}
& [0; \pi] \quad \rightsquigarrow \quad [1; \pi] \wedge [4; \pi] \\
\overset{2\times}{\rightsquigarrow}\ & (\neg pos(x) \to [2; \pi]) \wedge (pos(x) \to [5; \pi]) \\
\overset{2\times}{\rightsquigarrow}\ & (\neg pos(x) \to \{x := suc(x)\}[3; \pi]) \wedge (pos(x) \to (pos(x) \wedge [6; \pi])) \\
\overset{2\times}{\rightsquigarrow}\ & (\neg pos(x) \to \{x := suc(x)\}[0; \pi]) \wedge (pos(x) \to (pos(x) \wedge \text{false} \to [7; \pi]))
\end{aligned}$$

Since in the last step, the index 6 is outside the index range of π, $[6, \pi]$ is equivalent to false $\to [7, \pi]$ which is obviously true. $[3, \pi]$ is the same as $[0, \pi]$ and a loop is entered. The next section covers how we deal with such situations. This is a very simple example. In larger, more complex programs, one can learn more about the verification condition if one can interact during its generation.

4 Invariant Rules

The rewrite rules in Thm. 10 and Cor. 11 allow the symbolic execution of an unstructured program in a stepwise manner. If a program contains no loops, symbolic execution eventually results in a formula free of atomic program formulas. However, as soon as the program flow allows a statement to be executed more than once during the run of a program, these rules can no longer remove atomic program formulas entirely. A calculus for symbolic execution requires rules using loop invariants to resolve programs with loops. Such rules will, naturally, closely resemble invariant rules which are used to resolve loops in structured programs.

First, we give the simple version of an invariant rule. Then, a rule involving termination is defined and, finally, a rule which preserves more context information. The latter two can canonically be combined to a rule with termination and context preservation.

4.1 Program Modifications

In classic dynamic logic, the invariant rule introduces new proof goals on the loop body, i.e. on a program which is a strict subprogram of the original code. We are not able to reduce the code to a subset of statements in *USDL* since no restriction is imposed on the targets of goto statements and any statement, also outside the loop body, may be addressed.

We need, however, a means to reduce the number of traces of a loop body to one. This is achieved by inserting new statements into the program under inspection. The insertion is problematic, however, since index changes may make goto statements point to wrong targets afterwards. To compensate for this effect, we introduce an offset correction function off^{k}_{m} which increments the target indices by k if they lie above the insertion point m.

$$\mathit{off}^{k}_{m}(a) = \begin{cases} a & \text{if } a \le m \\ a + k & \text{otherwise} \end{cases}$$

We also apply off^{k}_{m} to statements. Here, it operates only on the target indices of goto statements and behaves like the identity function on all other statements.

Definition 13 (Statement insertion). *For programs $\pi, \tau \in \Pi$ and an arbitrary index $m \in \mathbb{N}$, the insertion $\pi \lhd (\tau, m) \in \Pi$ of τ into π at position m is defined as*

$$\big(\pi \lhd (\tau, m)\big)[i] = \begin{cases} \mathit{off}^{len(\tau)}_{m}(\pi[i]) & \text{for } i < m \\ \tau[i - m] & \text{for } m \le i < m + len(\tau) \\ \mathit{off}^{len(\tau)}_{m}(\pi[i - len(\tau)]) & \text{for } m + len(\tau) \le i \end{cases} .$$

τ is not subject to an offset correction since the programs we use for insertion here will not contain goto statements.

Fig. 2 shows a sample program insertion. The program $\tau = (\mathsf{assert}\ \phi; \mathsf{assume}\ \phi)$ is inserted into the program $\pi = (\mathsf{goto}\ 2; y := x; \mathsf{goto}\ 1)$ at position 1. Please note that in statement $4 : \mathsf{goto}\ 1$ of the resulting program, the target has *not* been incremented and still refers to the insertion point even though the statement to which it points has been changed.

Due to the index adaption off^{k}_{m}, a trace for π which does not pass through the insertion point m induces a trace for the program after insertion (of course with possibly adapted statement indices). The only way to enter the inserted statement sequence is to reach statement m, either as a goto target or by "walking" into it. Hence, if m is not part of the trace, we can observe:

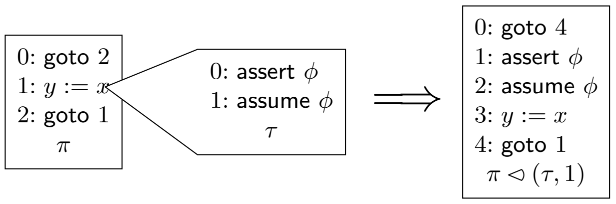

Fig. 2. Example of a program insertion

Property 14. For any trace $(I_0, k_0), \ldots, (I_r, k_r)$ with $k_i \neq n$ for $0 < i \leq r$, the sequence $(I_0, k'_0), \ldots, (I_r, k'_r)$ with $k'_i = \mathit{off}_n^{len(\tau)}$ is a trace for $\pi \lhd (\tau, n)$.

The rules we develop in this section are inference rules for a *sequent calculus*. A sequent is of the form $\Gamma \vdash \Delta$ with *antecedent* Γ and the *succedent* Δ finite sets of formulas. The sequent has the same truth value as the formula $(\bigwedge \Gamma) \to (\bigvee \Delta)$.

One problem that is not present in structured dynamic logic but with which we have to cope here, is the detection of loops. In classic dynamic logic, a loop can be identified syntactically as a statement initiated with the keyword "while". We do not have such landmarks in an unstructured program. A loop becomes a loop because of a goto statement targeting backward. Not every such statement, however, is necessarily an indicator for a loop. Therefore, we formulate our invariant rules in such a manner that they can be applied to *every* statement. Of course, the application is not equally expedient for all execution states, and it is the task of either a static analysis or the translation mechanism to identify (and to mark) the points at which an invariant rule should be applied.

4.2 Simple Invariant Rule

The general idea in the upcoming invariant rules is to change a program in such a way that a loop becomes dissected. At the beginning of the loop, an invariant is assumed which has to be asserted whenever the initial statement is reached again during symbolic execution. For that purpose we insert the statements (assert ϕ; assume false) at the current position.

Theorem 15. *The rule*

$$\frac{\Gamma \vdash \{\mathcal{U}\}\psi, \Delta \qquad \psi \vdash [n+2; \rho_1]}{\Gamma \vdash \{\mathcal{U}\}[n; \pi], \Delta}$$

with $\rho_1 = \pi \lhd ((\text{assert } \psi; \text{assume false}), n)$ *is a sound rule for any formula* ψ*.*

This rule has two premisses: The first provides evidence that the invariant ψ holds initially when arriving in the current state. The second premiss requires that in a state in which the invariant holds, the execution of the changed program is successful. Please note that the antecedent and succedent contexts Γ and Δ are not present in the second premiss. We will address this issue in Thm. 18.

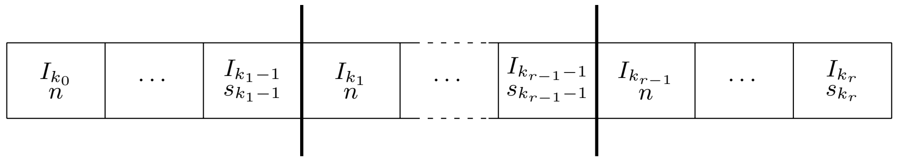

Fig. 3. Chopping a trace into subtraces

This rule is similar to the invariant rule for a dynamic logic for a simple 'while'-language. One difference is that, here, we have *two* rather than *three* premisses to establish. This is due to the fact that multiple assertions are embedded into the program ρ_1 and the second premiss $[n+2; \rho_1]$ plays two roles: It proves the absence of assertion violations after the loop (the 'use case' of ψ), and it ensures that the loop body preserves ψ establishing it as an invariant.

Proof. We can without loss of generality[1] assume that $\Delta = \emptyset$. Moreover, we may assume that (A) $\bigwedge\Gamma \to \{\mathcal{U}\}\psi$ and (B) $\psi \to [n+2; \rho_1]$ are valid formulas. For an arbitrary interpretation[2] I, we need to show that $I \models \bigwedge\Gamma \to \{\mathcal{U}\}[n; \pi]$. If $I \not\models \bigwedge\Gamma$, the proof is completed. Thus, let $I \models \bigwedge\Gamma$. It remains to be shown that $I \models \{\mathcal{U}\}[n; \pi]$. Setting $I_{k_0} := I^{\mathcal{U}}$ yields, equivalently, $I_{k_0} \models [n; \pi]$.

Let us look at an arbitrary maximal finite trace now. We can divide this trace in "loops to n", i.e., we split the trace into r subsequences such that every occurrence of n starts a new subtrace. For any $0 \le i < r$, the state (I_{k_i}, n) initiates a subtrace. The last trace ends in state (I_{k_r}, s_{k_r}). See Fig. 3 for an illustration.

We now claim that for every first state (I_{k_i}, n) of a subtrace, $I_{k_i} \models \psi$ holds and show this by induction on $0 \le i < r$. For $I_{k_0} (= I^{\mathcal{U}})$, this is a simple consequence of the validity of (A). Now, we assume that $I_{k_i} \models \psi$ for some $0 \le i < r-1$.

For the trace $(I_{k_i}, n), \dots, (I_{k_{i+1}-1}, s_{k_{i+1}-1})$, apart from the first state, no state is in statement n: it matches the requirements of Prop. 14, and, thus, we know that $(I_0, n+2), \dots, (I_{k_{i+1}-1}, \mathit{off}^2_n(s_{k_{i+1}-1}))$ is a trace for program ρ_1. From the original trace we know that $(I_{k_{i+1}}, n)$ is a successor state to the last state of this trace. Furthermore, $\rho_1[n] = \mathsf{assert}\ \psi$ and every maximal finite trace for ρ_1 is successful by assumption (B). This implies directly that the assert-condition is true, i.e. that $I_{k_{i+1}} \models \psi$.

We have seen now that every subtrace begins in an interpretation in which ψ holds. In particular, we have $I_{k_{r-1}} \models \psi$. The last subtrace $(I_{k_{r-1}}, n), \dots, (I_{k_r}, s_{k_r})$ is maximal (since the entire trace was chosen maximal). Statement n does not appear after the first state of this trace. We can therefore apply Prop. 14 again and obtain a trace $(I_{k_{r-1}}, n+2), \dots, (I_{k_r}, \mathit{off}^2_n(s_{k_r}))$ which is maximal again. Due to assumption (B), this trace must be successful, implying that the entire trace is successful. □

[1] There are first order inference rules that allow us to move the negation of all formulas in Δ to the antecedent Γ.
[2] For the sake of readability, we leave variable assignments aside in this section.

4.3 Invariant Rule with Termination

Thm. 15 is not sufficient if we want to incorporate the question of termination into the verification process. The rule for the terminating modality $[[\cdot]]$ introduces a variant term whose value strictly decreases from iteration to iteration. We assume there is a binary predicate symbol $\prec \in \mathrm{Prd}$ whose interpretation is a well-founded relation. With the aid of this predicate symbol, we can formulate an invariant rule which includes termination.

Theorem 16. *The rule*

$$\frac{\Gamma \vdash \{\mathcal{U}\}\psi, \Delta \qquad \psi \vdash \{nc := var\}[[n+2; \rho_2]]}{\Gamma \vdash \{\mathcal{U}\}[[n; \pi]], \Delta}$$

with $\rho_2 = \pi \lhd ((\mathsf{assert}\ \psi \wedge var \prec nc; \mathsf{assume}\ \mathrm{false}), n)$ *is a sound rule for any formula* ψ*, any term var, and a program variable nc which does not yet appear elsewhere on the sequent.*

Proof. Partial correctness $[n; \pi]$ is a direct consequence of Thm. 15 since we made the program modification *stronger* requiring $\psi \wedge var \prec nc$ to hold instead of only ψ.

Like in the proof above, we fix an interpretation I with $I \models \bigwedge \Gamma$ and set $I_{k_0} := I^{\mathcal{U}}$. It remains to be shown that there is no infinite trace for π starting in (I_{k_0}, n). Assuming there is such an infinite trace, we could subdivide it into subtraces such that every occurrence of the statement n initiates a new subtrace like in the previous proof. We can use the induction from the proof of Thm. 15 to establish that for every first state (I_{k_i}, n) of a subtrace we have $I_{k_i} \models \psi$.

In case there are finitely many subtraces, the last subtrace $\big((I_{k_{r-1}}, n), \dots\big)$ must be infinitely long and does not pass through n. We have $I_{k_{r-1}} \models \psi$ which already contradicts the second premiss which forbids an infinite trace for π starting in $(I_{k_{r-1}}, n)$ (because it uses the operator for total modality).

In case of infinitely many subtraces, every subtrace is finite. For the first states of the subtraces, we define $v_i := \mathrm{val}_{I_{k_i}}(var)$. If we take one beginning state (I_{k_i}, n) with $i > 0$, we know that (*) $I_{k_i} \models var \prec nc$ since this formula is part of the asserted loop invariant. As nc does not occur elsewhere on the sequents and because of the semantics of the update $nc := var$, we get that nc holds the value of var of the previous iteration, i.e. $I_{k_i}(nc) = v_{i-1}$. This and (*) imply that $(v_{i-1}, v_i) \in I(\prec)$. The sequence $(v_1, v_2, \dots)$ would therefore be an infinitely descending chain for $I(\prec)$ which cannot be since $\prec$ was chosen as a well-founded relation. $\square$

4.4 Improved Invariant Rule

The major disadvantage of the rules in Thms. 15 and 16 is that the information contained in Γ and Δ of the conclusion is not available in the second premiss. There invariant ψ is the only formula in the antecedent of the sequent. If any of the information encoded in $\Gamma \cup \Delta$ was needed to close the proof, it would have to be implied by ψ and one would need to proof its validity.

We will provide an invariant rule which keeps the context Γ and Δ but subjects those program variables which are touched during a loop iteration to a generalisation. We can use the havoc statement to do this generalisation because of (4).

The rule follows the ideas of [3] where a context preserving invariant rule is defined for a structured dynamic logic. The advantage is that more information on the sequent remains available and does not need to be encoded in the invariant.

Definition 17 (loop-reachable). *A statement m is called* loop-reachable *from n within a program π if there is a trace $(I_o, k_0), (I_1, k_1), \ldots$ such that*

1. *$k_o = n$,*
2. *there is an index $r \geq 1$ with $k_r = m$, and*
3. *there is an index $s > r$ with $k_s = n$.*

We denote this as $reach(n, m, \pi)$.

We use the notion of reachability to define the set of possibly modified program variables as

$$mod(n, \pi) := \left\{ c \;\middle|\; \begin{array}{l} \text{there are } m, c \text{ and } t \text{ s.t. } reach(n, m, \pi) \text{ and} \\ (\pi[m] = \mathsf{havoc}\ c \text{ or } \pi[m] = c := t) \end{array} \right\} \subseteq \mathrm{PVar} \quad .$$

Loop reachability can, in general, not be computed. The reachability of a statement may depend on the satisfiability of an assumption statement earlier in the execution path and this is undecidable. However, a static analysis can be used to over-approximate $mod(n, \pi)$.

The modified program ρ_3 is now more complex. The first two statements have the same intention as in Thm. 15 and the concluding assumption corresponds to the formula ψ in the antecedent of the second premiss in Thm. 15. The remaining statements need to be added to anonymise the values of those program variables that are possibly changed by the execution of the loop body.

Theorem 18. *The rule*

$$\frac{\Gamma \vdash \{\mathcal{U}\}\psi, \Delta \qquad \Gamma \vdash [n+2; \rho_3], \Delta}{\Gamma \vdash \{\mathcal{U}\}[n; \pi], \Delta}$$

with

$$\rho_3 \;=\; \pi \lhd ((\mathsf{assert}\ \ \psi; \mathsf{assume}\ \ \mathrm{false}; \mathsf{havoc}\ \ r_1; \ldots; \mathsf{havoc}\ \ r_b; \mathsf{assume}\ \ \psi), n)$$

is a sound rule for any formula ψ and any finite set $\{r_1, \ldots, r_b\}$ with $mod(n; \pi) \subseteq \{r_1, \ldots, r_b\} \subseteq \mathrm{PVar}$.

Proof. Again, let $\Delta = \emptyset$. We observe that the second premiss is (after a number of steps of symbolic execution and simplification) equivalent to

$$\Gamma \vdash \forall x_1. \ldots \forall x_b. \{r_1 := x_1 \| \ldots \| r_b := x_b\}(\psi \to [n + 2 + b + 1; \rho_3])$$

which by construction (the inserted havoc and following assume statements cannot be executed again) is equivalent to

$$\Gamma \vdash \forall x_1. \ldots \forall x_b. \{r_1 := x_1 \| \ldots \| r_b := x_b\}(\psi \to [n + 2; \rho_1]) \ . \qquad (9)$$

For an interpretation I with $I \models \bigwedge \Gamma$, we know, because of the validity of the premiss, that I makes the formula in (9) true. If an interpretation I' differs from I at most on the values of the program variables $r_1, \ldots, r_b$, then we have due to the semantics of the quantifier and the updates that also

$$I' \models (\psi \to [n + 2; \rho_1]) \ .$$

For a trace for $[n; \pi]$ (cf. Fig. 3) we observe that every statement before $(I_{k_{r-1}}, n)$ is loop-reachable from n. The program variables which are changed over this trace are, hence, in $mod(n, \pi)$ and, therefore, also among the $\{r_1, \ldots, r_b\}$. This implies that for all $0 \leq i < r$, the interpretation I_{k_i} coincides with I on the required program variables and we obtain $I_{k_i} \models (\psi \to [n + 2; \rho_1])$ and, hence, $I_{k_i} \models [n + 2; \rho_1]$ by induction from the proof of Thm. 15.

In particular we have $I_{k_{r-1}} \models [n + 2; \rho_1]$ for which we saw in the proof of Thm. 15 that it implies that the entire trace is successful. □

5 Related Work

While some verification tools (e.g., [2], [14]) take advantage of the greater transparency of source code verification, most employ a special-purpose intermediate language. The *Why* language [8] and the Forge Intermediate Representation (FIR) [7], for instance, are used as the target languages by various tools. Also, verification using the low level virtual machine (LLVM) format is a topic of ongoing research [13]. Boogie [6, 10] is the most popular intermediate language and is used as target language for various object-oriented and imperative source code languages (incl. C$^{\#}$, Java, Dafny, Eiffel, ...). Barnett and Leino [1] describe how the Boogie verification condition generator breaks up loops using invariants in a fashion similar to this work. In [12], Quigley defines a Hoare-style calculus for Java bytecode. It includes a loop rule which is similar to the inference rules of Sect. 4, but is more evolved due to the higher complexity of the Java bytecode. Burdy and Pavlova describe a wp-calculus for Java bytecode in [5]. Therein, loops are resolved by a code modification rendering the control flow acyclic prior to the wp-calculation. HOL/Boogie [4], like this work, aims for a combination of intermediate language and interactive verification. There, the generated verification conditions can be interactively proved; their generation, however, (i.e., the symbolic execution) remains inaccessible.

6 Conclusion

In this paper, we have presented a dynamic logic *USDL* for an unstructured verification language. The logic differs from Harel's logic as presented in [9] as it contains the formulas to be verified embedded in the program code. We have provided a model-theoretic semantics for *USDL* and calculus rules for the symbolic execution of programs within *USDL* formulas. For the treatment of loops, we have proved the soundness of three invariant rules.

The presented calculus has been implemented in an interactive, rule-based proof-of-concept tool which has been used to successfully conduct first experiments on the benefits of interaction in verification with intermediate languages.

Acknowledgements The author would like to thank Peter H. Schmitt for his constructive comments which helped improve this paper.

References

1. M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In M. D. Ernst and T. P. Jensen, editors, *PASTE 2005*, pages 82–87. ACM, 2005.
2. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.
3. B. Beckert, S. Schlager, and P. H. Schmitt. An improved rule for while loops in deductive program verification. In K.-K. Lau and R. Banach, editors, *ICFEM 05*, volume 3785 of *LNCS*, pages 315–329. Springer, 2005.
4. S. Böhme, K. R. M. Leino, and B. Wolff. HOL-Boogie—An interactive prover for the Boogie program-verifier. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *TPHOLs 2008*, volume 5170 of *LNCS*, pages 150–166. Springer, 2008.
5. L. Burdy and M. Pavlova. Java bytecode specification and verification. In L. M. Liebrock, editor, *SAC 2006*. ACM, 2006.
6. R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, Redmond, 2005.
7. G. Dennis, F. S.-H. Chang, and D. Jackson. Modular verification of code with SAT. In L. L. Pollock and M. Pezzè, editors, *ISSTA 2006*, pages 109–120. ACM, 2006.
8. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *CAV 2007*, volume 4590 of *LNCS*, pages 173–177. Springer, 2007.
9. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
10. K. R. M. Leino. This is Boogie 2, 2008. Manuscript KRML 178. Available at `http://research.microsoft.com/~leino/papers.html`.
11. K. R. M. Leino and P. Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In J. Esparza and R. Majumdar, editors, *TACAS 2010*, number 6015 in LNCS, pages 312–327. Springer, 2010.
12. C. L. Quigley. A programming logic for Java bytecode programs. In D. A. Basin and B. Wolff, editors, *TPHOLs 2003*, pages 41–54, 2003.
13. C. Sinz, S. Falke, and F. Merz. A precise memory model for low-level bounded model checking. In R. Huuck, G. Klein, and B. Schlich, editors, *SSV 2010*, 2010.
14. K. Stenzel. *Verification of Java Card Programs*. PhD thesis, University of Augsburg, 2005.