Dynamic Logic with Trace Semantics[⋆]

Bernhard Beckert and Daniel Bruns[⋆⋆]

Karlsruhe Institute of Technology (KIT), Germany

Abstract. Dynamic logic is an established instrument for program verification and for reasoning about the semantics of programs and programming languages. In this paper, we define an extension of dynamic logic, called Dynamic Trace Logic (DTL), which combines the expressiveness of program logics such as dynamic logic with that of temporal logic. And we present a sound and relatively complete sequent calculus for proving validity of DTL formulae.
Due to its expressiveness, DTL can serve as a basis for proving functional and information-flow properties in concurrent programs, among other applications.

1 Introduction

Overview. Dynamic logics (DL) [8] are multi-modal first-order logics where each legal sequential program fragment π (i.e., a sequence of statements) gives rise to a modal operator $[\pi]$. The formula $[\pi]\varphi$ expresses "in any state in which π terminates, φ holds." An interesting special case are deterministic programing languages, for which there is at most one terminal state. Program logics like DL are more expressive than Hoare logics in that programs are part of formulae, which can be self-composed. This allows, for instance, to express information-flow properties such as non-interference [12]. In other regards, however, standard dynamic logic lacks expressiveness: The semantics of a program is a relation between states; formulae can only describe the input/output behaviour of programs. It is inadequate for reasoning about non-terminating programs and for verifying temporal properties.

To combine the advantages of dynamic logic and temporal logic, our Dynamic Trace Logic uses trace-based program semantics and the well-known temporal operators $\Box$ (always), $\Diamond$ (eventually), $\bullet$ (weak next), $\circ$ (strong next), **U** (until), **W** (weak until), and **R** (release) similar to those of Linear Temporal Logic (LTL). In DTL, the formula $[\![\pi]\!]\varphi$ expresses that φ holds for the (possibly infinite) trace of the program π when started in the current state. For example, the formula

$$[\![\pi]\!]\Box\forall u.\forall v.(X \doteq u \land \circ(X \doteq v) \to u \leq v)$$

[⋆] This is the authors' version of the paper for uses defined by § 3 of the Springer-Verlag copyright terms. The final publication is available at `http://link.springer.com`.

[⋆⋆] This work has been supported by Deutsche Forschungsgemeinschaft (DFG) under project "Program-level Specification and Deductive Verification of Security Properties (DeduSec)" within SPP 1496 "Reliably Secure Software Systems (RS^3)".

is a two-state invariant. It says that the value of the program variable X must increase or remain the same throughout the trace of π. Proving such two-state invariants is the basis of the rely-guarantee approach for verifying concurrent programs.

Target Programing Language. In the following, we use a simple while language as target programing language without method calls or any feature of object-orientation. However, our language distinguishes between local variables with state-internal assignments and global variables with assignments inducing state transitions. The rationale behind this is that, in a concurrent setting, only global variables can be observed by the environment.

Of course, to be useful in practice, DTL needs to be extended to real-world programing languages. The KeY verification system (co-developed by the authors) is built on a calculus for JavaDL, a dynamic logic for sequential Java [3,5]. This has been used as a basis to extend DTL to Java and implement the DTL calculus (a prototypical implementation exists). Additional rules needed to handle full (sequential) Java can be derived from the KeY rules for the $[\cdot]$ modality by analogy. Since a language like Java incorporates a lot of features, in particular object-orientation, and various syntactic sugars, the rule set is rather voluminous in comparison to simple while languages. These special cases can, however, be reduced to a smaller set of base cases. For instance, the assignment `x=y++` containing a post-increment operator is transformed into two consecutive assignments `x=y` and `y=y+1` during symbolic execution.

Related Work. In earlier work [6], we have extended Dynamic Logic with a modality also written $[\![\cdot]\!]$, where $[\![\pi]\!]\varphi$ stands for "φ holds throughout the execution of π." This can be seen as a special case of DTL because the same property can be expressed in DTL as $[\![\pi]\!]\Box\varphi$. That is, in our earlier work, the temporal formula was restricted to the form $\Box\varphi$ with φ not containing further temporal operators. Platzer [10] introduced Temporal Dynamic Logic (dTL), where programs are *hybrid programs*; in particular, they are indeterministic, and therefore, traces are branching. It features formulae of the shapes $[\![\pi]\!]\Box\varphi$ ("for all traces, φ always holds") and $\langle\!\langle\pi\rangle\!\rangle\Diamond\varphi$ ("there is a trace such that eventually φ holds") where φ is a state formula. There is no further combination of temporal operators. Similar to our setting in this paper, traces can be of finite or infinite length. Platzer presents a sequent calculus for dTL, which, however, is incomplete, much due to the continuous state space of hybrid programs.

Reasoning about temporal properties is traditionally the domain of model checking. Nevertheless, there is some work on deductive techniques (tableaux, sequent calculi, resolution etc.) applied to temporal logics. Good sources on the topic of theorem proving for propositional linear-time logics are an article by Wolper [15] and the textbook chapters by Goré [7] and Reynolds and Dixon [11]. The work by Wolper introduces a tableau method for propositional LTL. A calculus for first-order LTL has been presented by Abadi and Manna [1]. It is known that, although LTL is decidable, there does not always exist a finite proof tree. The proof graph may contain cycles in the presence of eventualities (i.e., formulae with a positive occurrence of **U**). There are different techniques to deal with this. In the calculus presented in this paper, we use program invariants.

Language-based program verification is usually done w.r.t. state or two-state formula (pre and post). Program verification w.r.t. temporal specifications has been considered by Schellhorn et al. [13], where programs themselves are formulae of Interval Temporal Logic (ITL) [9]. In an earlier work, they have presented a sequent calculus for ITL [14], which allows to prove the correctness of programs w.r.t. ITL specifications.

Structure of this Paper. Syntax and semantics of our logic DTL are defined in Sects. 2 resp. 3 (including syntax and semantics of the while language that we use as the target programing language in this paper). In Sect. 4, we present our sequent calculus for DTL. Notions of soundness and completeness are defined in Sect. 5, and we sketch soundness and completeness proofs. Complete proofs can be found in an extended version of this paper [4].

2 Syntax of DTL

Signatures and Expressions. We assume disjoint sets *LVar* of local program variables and *GVar* of global program variables to be given. In addition, there is a set V of logical variables. Logical variables are rigid, i.e., they cannot be changed by programs and – in contrast to program variables – are assigned the same value in all states of a program trace. Quantifiers can only range over logical variables and not over program variables. In this paper, the sets of function and predicate symbols are fixed. They only contain the usual integer and boolean operators with their standard semantics.

Definition 1 (Expressions). Expressions *of type integer are constructed as usual over integer literals, local and global variables, logical variables, and the operators* $+$, $-$, $*$. *Expressions of type boolean are constructed using the relations* $\doteq$, $>$, $<$ *on integer expressions, the boolean literals true and false, and the logical operators* $\wedge$, $\vee$, $\neg$.

Programs. Programs are written in a simple while language, with the (mathematical) integers as the only data type. Expressions can be of types integer and boolean; they do not have side-effects. The program language does not contain features such as functions and arrays; and there are no object-oriented features. As discussed above, all such features can be added, but we keep the programing language simple for the presentation in this paper.

The only special feature is the distinction between local variables (written in lowercase letters) and global variables (written in uppercase). As will be explained in Sect. 3, we consider assignments to global variables to be the only program statements that lead to a new observable state. As a technical restriction, to ensure that there cannot be a program that gets stuck in an infinite loop without ever progressing to a new observable state, we demand that in every loop execution, an assignment to a global variable is executed.[1]

Definition 2 (Statements, programs). Programs *and* statements *are inductively defined, where statements are of the form:*

- `x = a;` *where* `x` $\in$ *LVar and* `a` *is an expression of type integer* (assignment to local variable)*,*
- `X = a;` *where* `X` $\in$ *GVar and* `a` *is an expression of type integer* (assignment to global variable)*,*
- `if (a) {`π_1`} else {`π_2`}` *where* `a` *is an expression of type boolean not containing logical variables and π_1 and π_2 are programs* (conditional)*, or*
- `while (a) {`π`}` *where* `a` *is an expression of type boolean not containing logical variables and π is a program that contains at least one assignment to a global variable on every execution path* (loop)*.*

Programs are finite sequences of statements. The empty program is denoted by ϵ.

State Updates. An important property of the calculus for DTL presented in Sect. 4 (as well as the calculus for JavaDL used in the KeY System) is that programs are *symbolically executed* starting from an initial state – in contrast to *wp*-calculi where one starts with a postcondition and works in a backwards manner. In order to capture the state transitions in between, we use a prefix on formulae, called *state update*. Updates can be thought of as "delayed substitutions," i.e., a substitution takes place once the program has been completely eliminated.

Definition 3 (State updates). *Let x be a (local or global) program variable, and let a be an expression. Then, $\{x := a\}$ is an update.*

For instance, $\{x := 4\}$ and $\{x := x + 1\}$ are updates. Applying these updates (after each other, from right to left) to the formula $x \doteq 5$ yields $4 + 1 \doteq 5$.

DTL Formulae. Formulae have the general appearance $\mathcal{U}[\![\pi]\!]\varphi$ where $\mathcal{U}$ is a sequence of updates, π is a program, and φ is a formula (that may or may not contain temporal operators and further sub-formulae of the same form). Intuitively, $\mathcal{U}[\![\pi]\!]\varphi$ expresses that φ holds when evaluated over all traces τ such that the initial state of τ is (partially) described by $\mathcal{U}$ and the further states of τ are constructed by running the program π.

Definition 4 (Formulae). State formulae *and* trace formulae *are inductively defined as follows:*

0. *All boolean expressions (Def. 1) are state formulae.*
1. *All state formulae are also trace formulae.*

[1] This property is undecidable in general, but a sufficient syntactical criterion could be that every possible execution path contains an assignment (which may be ineffective, e.g., `X = X;`).

2. *If φ and ψ are (state or trace) formulae, then the following are trace formulae: $\Box\varphi$ (always), $\bullet\varphi$ (weak next), $\varphi\,\mathbf{U}\,\psi$ (until).*
3. *If $\mathcal{U}$ is an update and φ a state formula, then $\mathcal{U}\varphi$ is a state formula.*
4. *If π is a program and φ a trace formula, then $[\![\pi]\!]\varphi$ is a state formulae.*
5. *The sets of state and trace formulae are closed under the logical operators $\neg, \wedge, \forall$.*

In addition, we use the following abbreviations:

$$\begin{aligned} \Diamond\varphi &:= \neg\Box\neg\varphi, & \circ\,\varphi &:= \neg\bullet\neg\varphi, \\ \varphi\,\mathbf{W}\,\psi &:= \varphi\,\mathbf{U}\,\psi \vee \Box\varphi, & \varphi\,\mathbf{R}\,\psi &:= \neg(\neg\varphi\,\mathbf{U}\,\neg\psi), \\ \varphi \vee \psi &:= \neg(\neg\varphi \wedge \neg\psi), & \varphi \to \psi &:= \neg\varphi \vee \psi, \\ \exists x.\varphi &:= \neg\forall x.\neg\varphi. \end{aligned}$$

A formula is called non-temporal *if it neither contains a temporal operator nor a program modality $[\![\pi]\!]$.*

3 Semantics of DTL

Expressions and formulae are evaluated over traces of states (which give meaning to program variables) and variable assignments (which give meaning to logical variables). The domain of DTL is always $\mathbb{Z}$, irregardless of the state (*constant domain*).

Definition 5 (States, variable assignments). *A* state *s is a function assigning integer values to all local and global variables, i.e., $s : LVar \cup GVar \to \mathbb{Z}$.*

A variable assignment *β is a function assigning integer values to all logical variables, i.e., $\beta : V \to \mathbb{Z}$.*

We use the notation $s\{x \mapsto d\}$ to denote the state that is identical to s except that the variable x is assigned the value $d \in \mathbb{Z}$. Likewise, we write $\beta\{x \mapsto d\}$ and $\tau\{x \mapsto d\}$ (where τ is a trace, see below) with the obvious semantics.

Definition 6 (Traces). *A* trace *τ is a non-empty, finite or infinite sequence of (not necessarily different) states.*

We use the following notations related to traces: (i) $|\tau| \in \mathbb{N}\cup\{\infty\}$ is the length of a trace τ. (ii) $\tau_1 \cdot \tau_2$ is the concatenation of traces. (iii) $\tau[i, j)$ for $i, j \in \mathbb{N} \cup \{\infty\}$ is the subtrace beginning in the i-th state (inclusive) and ending before the j-th state. (Indices out of bounds are treated as $\tau[0, j)$ or $\tau[i, |\tau|)$, respectively.) (iv) $\tau[i]$ for $i < |\tau|$ is the state at position i in τ.

Definition 7 (Semantics of expressions). *Given a state s and a variable assignment β, the value $a^{s,\beta}$ of an expression a in a state s is the integer or boolean value resulting from interpreting program variables x by x^s, logical variables u by u^β, and using the standard interpretation for all functions and relations.*

Program expressions that do not contain logical variables are independent of β, and we write a^s instead of $a^{s,\beta}$. If a is a boolean expression, we write $s, \beta \models a$ resp. $s \models a$ to denote that $a^{s,\beta}$ resp. a^s is true.

As mentioned in Sect. 2, we consider assignments to global variables to be the only statements that lead to a new observable state. By specifying which variables are local and which are global, the user can thus determine which states are "interesting" and are to be included in a trace.

For the feasibility of proving DTL formulae, it is important that not too many irrelevant intermediate states are included in a trace because, if a formula such as $[\![\pi]\!]\Box\varphi$ is to be proven valid, intermediate states require sub-proofs showing that φ holds in each of them.

Definition 8 (Trace of a program). *Given an initial state s, the* trace of a program π*, denoted $trc(s, \pi)$, is defined by (the greatest fixpoint of):*

$$\begin{aligned}
trc(s, \epsilon) &= \langle s\rangle \\
trc(s, \texttt{x = a;}\ \omega) &= trc(s\{x \mapsto a^s\}, \omega) \\
trc(s, \texttt{X = a;}\ \omega) &= \langle s\rangle \cdot trc(s\{X \mapsto a^s\}, \omega) \\
trc(s, \texttt{if (a) \{}\pi_1\texttt{\} else \{}\pi_2\texttt{\}}\ \omega) &= \begin{cases} trc(s, \pi_1\ \omega) & \text{if } s \vDash a \\ trc(s, \pi_2\ \omega) & \text{if } s \nvDash a \end{cases} \\
trc(s, \texttt{while (a) \{}\pi\texttt{\}}\ \omega) &= \begin{cases} trc(s, \pi\ \texttt{while (a) \{}\pi\texttt{\}}\ \omega) & \text{if } s \vDash a \\ trc(s, \omega) & \text{if } s \nvDash a \end{cases}
\end{aligned}$$

where ϵ is the empty program and ω is a program.

We have now everything needed to define the semantics of DTL formulae in a straightforward way. The valuation of a formula is given w.r.t. a trace τ and a variable assignment β. This is expressed by the validity relation, denoted by $\vDash$.

Definition 9 (Semantics of formulae). *Let τ be a trace and β a variable assignment. The* validity relation *is the smallest relation satisfying the following.*

$$\begin{array}{ll}
\tau, \beta \vDash a & \textit{iff } a^{\tau[0],\beta} = \textit{true} \\
 & \qquad \textit{(in case } a \textit{ is an expression, see Def. 7)} \\
\tau, \beta \vDash \neg\varphi & \textit{iff } \tau, \beta \nvDash \varphi \\
\tau, \beta \vDash \varphi \wedge \psi & \textit{iff } \tau, \beta \vDash \varphi \textit{ and } \tau, \beta \vDash \psi \\
\tau, \beta \vDash \forall u.\varphi & \textit{iff for every } d \in \mathbb{Z}\textit{: } \tau, \beta\{u \mapsto d\} \vDash \varphi \\
\tau, \beta \vDash \Box\varphi & \textit{iff } \tau[i, \infty), \beta \vDash \varphi \textit{ for every } i \in [0, |\tau|) \\
\tau, \beta \vDash \varphi\, \mathbf{U}\, \psi & \textit{iff } \tau[j, i), \beta \vDash \varphi \textit{ and } \tau[i, \infty), \beta \vDash \psi \\
 & \qquad \textit{for some } i \in [0, |\tau|) \textit{ and all } j \in [0, i) \\
\tau, \beta \vDash \bullet\,\varphi & \textit{iff } \tau[1, \infty), \beta \vDash \varphi \textit{ or } |\tau| = 1 \\
\tau, \beta \vDash \{x := a\}\varphi & \textit{iff } \tau\{x \mapsto a^{\tau[0]}\}, \beta \vDash \varphi \\
\tau, \beta \vDash [\![\pi]\!]\varphi & \textit{iff } trc(\tau[0], \pi), \beta \vDash \varphi
\end{array}$$

A formula φ is valid *if $\tau, \beta \vDash \varphi$ for all τ and all β.*

4 A Sequent Calculus for DTL

In this section, we present a sequent calculus for DTL, which we call $\mathcal{C}_{\mathrm{DTL}}$. It is sound and relatively complete, i.e., complete up to the handling of arithmetic (see Sect. 5). The calculus consists of the following rule classes:

Classical logic rules These rules simplify formulae whose top-level operator is a quantifier or a propositional operator.

Simplification and normalization rules Rules for simplifying formulae of the form $\mathcal{U}[\![\pi]\!]\varphi$, where the top-level operator in φ is not temporal.

Rules for temporal operators Rules that apply to formulae $\mathcal{U}[\![\pi]\!]\varphi$ with a top-level temporal operator in φ, and that do not change the program π.

Program rules Rules that apply to formulae of the form $\mathcal{U}[\![\pi]\!]\varphi$, and that analyze and/or simplify the program π. Not surprisingly, this class has the most complex rules, including invariant rules for loops.

Rules for data structures Since our focus in this paper is not on how to handle arithmetics, we use oracle rules for arithmetics.

Other rules This category includes the closure and the cut rule.

Most rules of the calculus are analytic and therefore can be applied automatically. The rules that require user interaction are: (a) the rules for handling while loops (where a loop invariant has to be provided), (b) the cut rule (where the right case distinction has to be used), and (c) the quantifier rules (where the right instantiation has to be found).

Traces are uniquely determined by symbolic program executions of the deterministic programing language. The general idea behind our calculus is to explore a trace until it terminates or reaches a fixpoint (induced by a non-terminating loop). Thus, proofs usually consist of alternating applications of temporal logic rules (which decompose trace formulae, e.g., $\Box\varphi$ to $\bullet\Box\varphi \wedge \varphi$) and program rules (which let us step forward in the trace). Those steps are explicitly given through assignments in the program.

In the rule schemata, Γ, Δ denote arbitrary, possibly empty multi-sets of formulae, φ, ψ denote arbitrary formulae, $\mathcal{U}$ stands for a (possibly empty) sequence of updates, π, ω for programs, γ is a state formula, `x` and `X` are local and global program variables, n and u are logical variables, a is an expression of type integer, and b is an expression of type boolean.

As usual, the schematic sequents above the horizontal line in a schema are its *premisses* and the single schematic sequent below the horizontal line is its *conclusion.* Note, that in practice the rules are applied from bottom to top. Proof construction starts with the original proof obligation at the bottom. Therefore, if a constraint is attached to a rule that requires a variable to be "new," it has to be new w.r.t. the conclusion.

Definition 10 (Soundness, derivability).

1. *A sequent* $\Gamma \vdash \Delta$ *is* valid *(in state* s *and under variable assignment* β*) if and only if the formula* $\bigwedge_{\gamma\in\Gamma} \gamma \to \bigvee_{\delta\in\Delta} \delta$ *is valid (w.r.t.* s, β*).*
2. *A rule* $\dfrac{\Gamma_1 \vdash \Delta_1 \quad \cdots \quad \Gamma_n \vdash \Delta_n}{\Gamma_0 \vdash \Delta_0}$ *is* sound *if, for all valid instances of premisses* $\Gamma_i \vdash \Delta_i$*, also the instance of* $\Gamma_0 \vdash \Delta_0$ *is valid.*
3. *A sequent is* derivable *(with* $\mathcal{C}_{\mathrm{DTL}}$*) if it is an instance of the conclusion of a rule schema and all corresponding instances of the premisses of that rule schema are derivable sequents. In particular, all sequents are derivable that*

Table 1. Rules for quantifiers, propositional operators, and state updates. In rule R5, the substitution needs to be admissible; rule R6 introduces a fresh variable u'. Rules R7 and R8 make use of weak substitution (Def. 12).

$$\frac{\Gamma \vdash \varphi, \Delta}{\Gamma, \neg\varphi \vdash \Delta}\ \text{R1} \qquad \frac{\Gamma, \varphi \vdash \Delta}{\Gamma \vdash \neg\varphi, \Delta}\ \text{R2}$$

$$\frac{\Gamma, \varphi, \psi \vdash \Delta}{\Gamma, \varphi \wedge \psi \vdash \Delta}\ \text{R3} \qquad \frac{\Gamma \vdash \varphi, \Delta \qquad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \varphi \wedge \psi, \Delta}\ \text{R4}$$

$$\frac{\Gamma, \varphi[u/a], \forall u.\varphi \vdash \Delta}{\Gamma, \forall u.\varphi \vdash \Delta}\ \text{R5} \qquad \frac{\Gamma \vdash \varphi[u/u'], \Delta}{\Gamma \vdash \forall u.\varphi, \Delta}\ \text{R6}$$

$$\frac{\Gamma, \mathcal{U}\varphi[x\sharp a] \vdash \Delta}{\Gamma, \mathcal{U}\{x := a\}\varphi \vdash \Delta}\ \text{R7} \qquad \frac{\Gamma \vdash \mathcal{U}\varphi[x\sharp a], \Delta}{\Gamma \vdash \mathcal{U}\{x := a\}\varphi, \Delta}\ \text{R8}$$

are instances of the conclusion of a rule that has no premisses (rules R22, R31, and R33).

4.1 Classical Logic and Update Rules

The rules for quantifiers, propositional operators, and updates are shown in Table 1. Note that the expressions that are used to instantiate universal quantifiers in rule R5 must be chosen in such a way that the substitution is admissible:

Definition 11 (Admissible substitution). *A substitution u/a of a logical variable $u \in V$ with an expression a is* admissible *w.r.t. a formula φ if there is no variable v in a such that u is free in φ and, after replacing a for some free occurrence of u in φ, the occurrence of v in a is* (i) *bound by a quantifier in $\varphi[u/a]$ (in case v is a logical variable) or is* (ii) *in the scope of a program modality $[\![\pi]\!]$ that contains an assignment to v (in case v is a program variable).*

For example, using `X` to instantiate the universal quantifier in the DTL formula $\forall u.(u \doteq 0 \rightarrow [\![\texttt{X = 1;}]\!]\Box u \doteq 0)$ is not admissible. Indeed the result would be incorrect as the original formula is valid while $X \doteq 0 \rightarrow [\![\texttt{X = 1;}]\!]\Box X \doteq 0$ is not even satisfiable. In order to deal with updates, we introduce the notion of *weak substitutions*, which avoid such clashes by definition.

Definition 12 (Weak substitution). *For a state formula φ and an update $\{x := a\}$ define the formula $\varphi[x\sharp a]$ according to the following schema:* (i) *if φ is an expression, then $\varphi[x\sharp a] = \varphi[x/a]$,* (ii) *if φ begins with an update or a program modality, then $\varphi[x\sharp a] = \{x := a\}\varphi$,* (iii) *if φ is a propositional junction, then the weak substitution is propagated, e.g., $(\varphi_1 \wedge \varphi_2)[x\sharp a] = \varphi_1[x\sharp a] \wedge \varphi_2[x\sharp a]$,* (iv) *if φ begins with a quantifier, then the weak substitution is propagated (possibly under renaming the bound variable so that it does not occur in a).*

4.2 Simplification and Normalization Rules

As said above, our calculus contains simplification rules that apply to formulae of the form $\mathcal{U}[\![\pi]\!]\varphi$, where the top-level operator in φ is not temporal. They are

Table 2. Simplification and normalization rules. In rule R16, γ is a state formula. Rule R17 introduces a fresh variable u'; in rule R18, the substitution needs to be admissible.

$$\frac{\Gamma \vdash \mathcal{U}\llbracket\pi\rrbracket\varphi,\ \mathcal{U}\llbracket\pi\rrbracket\psi, \Delta}{\Gamma \vdash \mathcal{U}\llbracket\pi\rrbracket(\varphi \vee \psi), \Delta}\ \text{R9} \qquad \frac{\Gamma \vdash \mathcal{U}\llbracket\pi\rrbracket\varphi, \Delta \qquad \Gamma \vdash \mathcal{U}\llbracket\pi\rrbracket\psi, \Delta}{\Gamma \vdash \mathcal{U}\llbracket\pi\rrbracket(\varphi \wedge \psi), \Delta}\ \text{R10}$$

$$\frac{\Gamma \vdash \mathcal{U}\llbracket\pi\rrbracket\neg\varphi, \Delta}{\Gamma \vdash \neg\mathcal{U}\llbracket\pi\rrbracket\varphi, \Delta}\ \text{R11} \qquad \frac{\Gamma \vdash \mathcal{U}\llbracket\pi\rrbracket\Box\neg\psi,\ \mathcal{U}\llbracket\pi\rrbracket(\neg\psi\ \mathbf{U}\ (\neg\varphi \wedge \neg\psi)), \Delta}{\Gamma \vdash \mathcal{U}\llbracket\pi\rrbracket\neg(\varphi\ \mathbf{U}\ \psi), \Delta}\ \text{R12}$$

$$\frac{\Gamma \vdash \mathcal{U}\llbracket\pi\rrbracket\neg\varphi, \Delta}{\Gamma, \mathcal{U}\llbracket\pi\rrbracket\varphi \vdash \Delta}\ \text{R13} \qquad \frac{\Gamma \vdash \mathcal{U}\llbracket\pi\rrbracket\varphi, \Delta}{\Gamma \vdash \mathcal{U}\llbracket\pi\rrbracket\neg\neg\varphi, \Delta}\ \text{R14}$$

$$\frac{\Gamma \vdash \mathcal{U}\llbracket\pi\rrbracket\circ\neg\varphi, \Delta}{\Gamma \vdash \mathcal{U}\llbracket\pi\rrbracket\neg\bullet\varphi, \Delta}\ \text{R15} \qquad \frac{\Gamma \vdash \mathcal{U}\gamma, \Delta}{\Gamma \vdash \mathcal{U}\llbracket\pi\rrbracket\gamma, \Delta}\ \text{R16}$$

$$\frac{\Gamma \vdash \mathcal{U}\llbracket\pi\rrbracket\varphi[u/u'], \Delta}{\Gamma \vdash \mathcal{U}\llbracket\pi\rrbracket\forall u.\varphi, \Delta}\ \text{R17} \qquad \frac{\Gamma \vdash \mathcal{U}\llbracket\pi\rrbracket\varphi[u/a],\ \mathcal{U}\llbracket\pi\rrbracket\exists u.\varphi, \Delta}{\Gamma \vdash \mathcal{U}\llbracket\pi\rrbracket\exists u.\varphi, \Delta}\ \text{R18}$$

shown in Table 2. In particular, they include normalization rules which deal with negated trace formulae through replacement by the respective dual formula.

Rule R12 for negated until avoids introducing the dual $\mathbf{R}$ into the sequent. Therefore, no rules for $\mathbf{R}$ are required in the calculus. Soundness of R12 follows from the well-known equivalence $\varphi\,\mathbf{R}\,\psi \leftrightarrow \psi\,\mathbf{W}\,(\varphi \wedge \psi)$ in LTL and the definitions of $\mathbf{R}$ and $\mathbf{W}$, which applies to finite traces as well (cf., e.g., [2]).

Since (for conciseness of the calculus) we only include program and temporal logic rules for the right-hand side of a sequent, we need rule R13 that allows to move a formula with a modality from the left of a sequence to the right.

In case φ is a state formula, rule R16 can be used to remove the program modality (as a state formula is evaluated in the initial state of a trace). Further simplification rules are applied to split formulae such as $\llbracket\pi\rrbracket(\Box\varphi \wedge \psi)$.

4.3 Rules for Temporal Operators

Table 3 shows the rules that handle temporal operators without changing the program. Rules R19 to R21 "unwind" temporal formulae by splitting them into a "future" part and a "present" part. Rules R22 and R23 handle the case of an empty program (i.e., empty remaining trace) for weak and strong next, respectively. Rule R22 also closes a proof branch.

Table 3. Rules for handling temporal operators

$$\frac{\Gamma \vdash \mathcal{U}(\llbracket\pi\rrbracket\circ(\varphi\ \mathbf{U}\ \psi) \wedge \llbracket\pi\rrbracket\varphi),\ \mathcal{U}\llbracket\pi\rrbracket\psi, \Delta}{\Gamma \vdash \mathcal{U}\llbracket\pi\rrbracket\varphi\ \mathbf{U}\ \psi, \Delta}\ \text{R19} \qquad \frac{\Gamma \vdash \mathcal{U}(\llbracket\pi\rrbracket\bullet\Box\varphi \wedge \llbracket\pi\rrbracket\varphi), \Delta}{\Gamma \vdash \mathcal{U}\llbracket\pi\rrbracket\Box\varphi, \Delta}\ \text{R20}$$

$$\frac{\Gamma \vdash \mathcal{U}\llbracket\pi\rrbracket\circ\Diamond\varphi,\ \mathcal{U}\llbracket\pi\rrbracket\varphi, \Delta}{\Gamma \vdash \mathcal{U}\llbracket\pi\rrbracket\Diamond\varphi, \Delta}\ \text{R21} \qquad \frac{}{\Gamma \vdash \mathcal{U}\llbracket\rrbracket\bullet\varphi, \Delta}\ \text{R22} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \mathcal{U}\llbracket\rrbracket\circ\varphi, \Delta}\ \text{R23}$$

4.4 Program Rules

The program rules are shown in Table 4. Assignments to local and global variables are handled by the rules R24 and R26, respectively. The former can be applied on any formula φ, while the latter one, which handles assignments to global variables, steps to the next state and consumes a (weak or strong) next operator.

Table 4. Program rules. The schematic symbol $\overset{\bullet}{\circ}$ stands for $\bullet$ or $\circ$.

$$\frac{\Gamma \vdash \mathcal{U}\{x := a\}\llbracket\omega\rrbracket\varphi, \Delta}{\Gamma \vdash \mathcal{U}\llbracket \mathtt{x = a;}\ \omega\rrbracket\varphi, \Delta}\ \mathsf{R24} \qquad \frac{\Gamma, \mathcal{U}b \vdash \mathcal{U}\llbracket\pi_1\ \omega\rrbracket\varphi, \Delta \qquad \Gamma, \mathcal{U}\neg b \vdash \mathcal{U}\llbracket\pi_2\ \omega\rrbracket\varphi, \Delta}{\Gamma \vdash \mathcal{U}\llbracket \mathtt{if\ (b)\ \{}\pi_1\mathtt{\}\ else\ \{}\pi_2\mathtt{\}}\ \omega\rrbracket\varphi, \Delta}\ \mathsf{R25}$$

$$\frac{\Gamma \vdash \mathcal{U}\{X := a\}\llbracket\omega\rrbracket\varphi, \Delta}{\Gamma \vdash \mathcal{U}\llbracket \mathtt{X = a;}\ \omega\rrbracket\overset{\bullet}{\circ}\varphi, \Delta}\ \mathsf{R26} \qquad \frac{\Gamma \vdash \mathcal{U}\llbracket \mathtt{if\ (b)\{}\pi\ \mathtt{while\ (b)\ \{}\pi\mathtt{\}\}\ else\ \{\}}\ \omega\rrbracket\varphi, \Delta}{\Gamma \vdash \mathcal{U}\llbracket \mathtt{while\ (b)\ \{}\pi\mathtt{\}}\ \omega\rrbracket\varphi, \Delta}\ \mathsf{R27}$$

An `if` statement is handled by splitting the formula in two parts, each containing the alternative program and the remaining program code as shown in rule R25. Similarly, loops can be handled by unwinding, as shown in rule R27. In the case in which the loop condition holds, the loop body is symbolically executed and then again the whole loop. In the second case where the loop condition does not hold, the loop is simply skipped. However, the number of loop iterations may not be known in advance, or the loop may not even terminate. In those cases, we need invariants.

Invariant rules are an established technique for handling loops in calculi for program logics. The basic idea is to have a state formula γ (the invariant) which holds in all states before and – if it terminates – after an execution of the loop body. If we can show that preservation, it only remains to show that φ holds on the remaining trace. The rules are shown in Table 5.

For a trace formula of the shape $\Box\varphi$, the four premisses of R28 intuitively state that (i) γ holds in the beginning; (ii) it is preserved by each loop iteration (i.e., it actually is an invariant), here a possible post-π state is characterized

Table 5. Invariant rules

$$\frac{\begin{array}{c}\Gamma \vdash \mathcal{U}\gamma, \Delta \qquad \gamma, b \vdash \llbracket\pi\rrbracket\Box(\bullet \mathit{false} \to \gamma) \qquad \gamma \vdash b, \llbracket\omega\rrbracket\Box\varphi \\ \gamma, b \vdash \llbracket\pi \mid \mathtt{while\ (b)\ \{}\pi\mathtt{\}}\ \omega\rrbracket\varphi\end{array}}{\Gamma \vdash \mathcal{U}\llbracket \mathtt{while\ (b)\ \{}\pi\mathtt{\}}\ \omega\rrbracket\Box\varphi, \Delta}\ \mathsf{R28}$$

$$\frac{\begin{array}{c}\Gamma \vdash \exists u.(u \geq 0 \land \mathcal{U}\mathcal{V}_u\gamma), \Delta \qquad n \geq 0 \vdash \mathcal{V}_{n+1}(\gamma \to (b \land \llbracket\pi\rrbracket\Diamond(\bullet \mathit{false} \land \mathcal{V}_n\gamma))) \\ \vdash \mathcal{V}_0(\gamma \to \llbracket \mathtt{while\ (b)\ \{}\pi\mathtt{\}}\ \omega\rrbracket\Diamond\varphi)\end{array}}{\Gamma \vdash \mathcal{U}\llbracket \mathtt{while\ (b)\ \{}\pi\mathtt{\}}\ \omega\rrbracket\Diamond\varphi, \Delta}\ \mathsf{R29}$$

$$\frac{\begin{array}{c}\Gamma \vdash \exists u.(u \geq 0 \land \mathcal{U}\mathcal{V}_u\gamma), \Delta \qquad n \geq 0 \vdash \mathcal{V}_{n+1}(\gamma \to (b \land \llbracket\pi\rrbracket\Diamond(\bullet \mathit{false} \land \mathcal{V}_n\gamma))) \\ \vdash \mathcal{V}_0(\gamma \to \llbracket \mathtt{while\ (b)\ \{}\pi\mathtt{\}}\ \omega\rrbracket\varphi_1\ \mathbf{U}\ \varphi_2) \qquad n > 0 \vdash \mathcal{V}_n(\gamma \to \llbracket\pi \mid \mathtt{while\ (b)\ \{}\pi\mathtt{\}}\ \omega\rrbracket\varphi_1)\end{array}}{\Gamma \vdash \mathcal{U}\llbracket \mathtt{while\ (b)\ \{}\pi\mathtt{\}}\ \omega\rrbracket\varphi_1\ \mathbf{U}\ \varphi_2, \Delta}\ \mathsf{R30}$$

by the temporal formula $\bullet \mathit{false}$; (iii) if the loop terminates, indicated by the negated loop condition b, then $\Box\varphi$ holds on the remaining trace; and (iv) for every loop iteration, φ holds throughout, i.e., for the remaining trace from every state during loop iterations. As an invariant abstracts from concrete loop iterations, the context Γ, Δ must be discarded in the all but the first premiss.

Note that – in contrast to invariant rules in state-based dynamic logic – it is not sound in premiss (iv), to decompose the program trace and to only regard the subtrace induced by π in isolation, i.e., just proving $[\![\pi]\!]\Box\varphi$ is not sound. As an example, consider the formula $[\![\texttt{while (X>0) \{X = X-1;\}}]\!]\Box\bullet\bullet\mathit{false}$, which is not valid, but the formula $[\![\texttt{X = X-1;}]\!]\Box\bullet\bullet\mathit{false}$, containing the loop body, obviously is. This means for a sound rule, that we have to consider the remaining trace as well. However, we are only interested in those traces which begin in the subtrace induced by the loop body π.

For this reason, we introduced another, two-place program modality: $[\![\pi \mid \omega]\!]\varphi$ means that for any state in the subtrace induced by π, trace formula φ holds for the remaining trace including ω. More formally, we define $[\![\pi \mid \omega]\!]\varphi$ as a shorthand for $[\![\texttt{x = 0;}\ \pi\ \texttt{x = 1;}\omega]\!](\varphi\, \mathbf{W}\, x \doteq 1)$ where local program variable x does not occur in π, ω, or φ. Even though the resulting formula is syntactically longer here, it is easier to prove in the sense that there are fewer states in which φ has to hold.

In the case of R29 ("diamond") and R30 ("until"), the invariant is accompanied by a sequence of updates $\mathcal{V}_u$ with an integer expression u, which describes the progress made through each loop iteration. The general shape of $\mathcal{V}_u$ is $\{x_1 := f_1(u)\} \cdots \{x_k := f_k(u)\}$ where $x_1, \ldots, x_k$ are variables appearing in γ and $f_1, \ldots, f_k$ are functions. The intuition behind it is that $\mathcal{V}_0\gamma$ describes either a state in which the loop terminates immediately or a fixpoint of the loop. Such a state must be reached in a finite number of iterations, which is guaranteed since n is decreasing in every iteration. For this reason, premiss (ii) requires executions of the loop body to terminate. In Rule R30, there is a fourth premiss stating that φ_1 holds throughout the loop body for every iteration where $n > 0$.

4.5 Rules for Data Structures

Our calculus is basically independent of the domain of computation resp. data structures that are used. We therefore abstract from the problem of handling the data structure(s) and just assume that an oracle is available that can decide the validity of non-temporal formulae in the domain of computation (note that the oracle only decides pure first-order formulae). In the case of arithmetic, the oracle is represented by rule R31 in Table 6.

Of course, the non-temporal formulae that are valid in arithmetic are not even enumerable. Therefore, in practice, the oracle can only be approximated, and rule R31 must be replaced by a rule (or set of rules) for computing resp. enumerating a *subset* of all valid non-temporal formulae (in particular, these rules must include equality handling). This is not harmful to "practical completeness." Rule sets for arithmetic are available, which – as experience shows – allow to derive all valid non-temporal formulae that occur during the verification of actual

Table 6. Oracle rules and induction rule for handling arithmetic (n is fresh)

$$\text{if } \bigwedge \Gamma \to \bigvee \Delta \text{ is a valid non-temporal formula: } \frac{}{\Gamma \vdash \Delta}\ \mathsf{R31}$$

$$\frac{\Gamma \vdash \varphi(0), \Delta \qquad \Gamma, \varphi(u) \vdash \varphi(u+1), \Delta}{\Gamma \vdash \forall u.\varphi(u), \Delta}\ \mathsf{R32}$$

Table 7. The closure and the cut rule

$$\frac{}{\Gamma, \varphi \vdash \varphi, \Delta}\ \mathsf{R33} \qquad \frac{\Gamma, \varphi \vdash \Delta \qquad \Gamma \vdash \varphi, \Delta}{\Gamma \vdash \Delta}\ \mathsf{R34}$$

programs. Using powerful SMT solvers, this can be done fully automatically in many cases. Typically, an approximation of the computation domain oracle contains a rule for structural induction. In the case of arithmetic, that is rule R32. This rule, however, not only applies to non-temporal formulae but also to DTL formulae containing programs.

The remaining rules, which are shown in Table 7, are the cut rule R34 (with an arbitrary cut formula φ) and the closure rule R33 which closes a proof branch.

5 Soundness and Completeness

Soundness of the calculus $\mathcal{C}_{\mathrm{DTL}}$ (Corollary 1) is based on the following theorem, which states that all rules preserve validity of the derived sequents.

Theorem 1. *For all rule schemata of the calculus $\mathcal{C}_{\mathrm{DTL}}$, R1 to R34, the following holds: If all premisses of a rule schema instance are valid sequents, then its conclusion is a valid sequent.*

Corollary 1. *If a sequent $\Gamma \vdash \Delta$ is derivable with the calculus $\mathcal{C}_{\mathrm{DTL}}$, then it is valid, i.e., $\bigwedge \Gamma \to \bigvee \Delta$ is a valid formula.*

Proving Theorem 1 is not difficult. The proof is, however, quite large as soundness has to be shown separately for each rule; the proof is given in [4, App. A].

The calculus $\mathcal{C}_{\mathrm{DTL}}$ is *relatively* complete; that is, it is complete up to the handling of the domain of computation (the data structures). It is complete if an oracle rule for the domain is available – in our case the oracle rule for arithmetic, R31. If the domain is extended with other data types, $\mathcal{C}_{\mathrm{DTL}}$ remains relatively complete; and it is still complete if rules for handling the extended domain of computation are added.

Theorem 2. *If a sequent is valid, then it is derivable with $\mathcal{C}_{\mathrm{DTL}}$.*

Corollary 2. *If φ is a valid DTL formula, then the sequent $\vdash \varphi$ is derivable.*

Due to space restrictions, the proof of Theorem 2, which is quite complex, cannot be given here. The basic idea of the proof is the same as that used by Harel [8] to prove relative completeness of his sequent calculus for first-order DL. An extensive proof sketch can be found in [4, App. B]. The following lemma is central to the completeness proof.

$$
\begin{array}{cl}
\hline
\vdash \{X := 5\}[\![]\!]\circ\Diamond X \geq 4,\ 5 \geq 4,\ [\![\texttt{X=5;}]\!]X \geq 4 & \mathsf{R31} \\
\hline
\vdash \{X := 5\}[\![]\!]\circ\Diamond X \geq 4,\ \{X := 5\}X \geq 4,\ [\![\texttt{X=5;}]\!]X \geq 4 & \mathsf{R8} \\
\hline
\vdash \{X := 5\}[\![]\!]\circ\Diamond X \geq 4,\ \{X := 5\}[\![]\!]X \geq 4,\ [\![\texttt{X=5;}]\!]X \geq 4 & \mathsf{R16} \\
\hline
\vdash \{X := 5\}[\![]\!]\Diamond X \geq 4,\ [\![\texttt{X=5;}]\!]X \geq 4 & \mathsf{R21} \\
\hline
\vdash [\![\texttt{X=5;}]\!]\circ\Diamond X \geq 4,\ [\![\texttt{X=5;}]\!]X \geq 4 & \mathsf{R26} \\
\hline
\vdash [\![\texttt{X=5;}]\!](\circ\Diamond X \geq 4 \vee X \geq 4) & \mathsf{R9} \\
\hline
\vdash [\![\texttt{X=5;}]\!]\Diamond X \geq 4 & \mathsf{R21}
\end{array}
$$

Fig. 1. Example proof tree (rules focus on the solid black formulae)

Lemma 1. *For every DTL formula φ_{DTL} there is an (arithmetical) non-temporal first-order formula φ_{FOL} that is logically equivalent to φ_{DTL}, i.e., for all traces τ and variable assignments β:*

$$\tau, \beta \vDash \varphi_{DTL} \quad \textit{iff} \quad \tau, \beta \vDash \varphi_{FOL} \ .$$

The above lemma states that DTL is not more expressive than first-order arithmetic. This holds as arithmetic – our domain of computation – is expressive enough to encode the behaviour of programs. In particular, using Gödelization, arithmetic allows to encode program states (i.e., the values of all the variables occurring in a program) and finite (sub-)traces into a single number. Further it is then possible to construct, for every DTL formula ψ, state s, program π, and $n \in \mathbb{N}$, a FOL formula $\varphi_{\psi,s,\pi,n}$ encoding that $\mathit{trc}(s, \pi)[n, \infty) \vDash \psi$.

Note that Lemma 1 states a property of the logic DTL that is independent of any calculus. It implies that a DTL formula could be decided by constructing an equivalent non-temporal formula and then invoking the computation domain oracle – if such an oracle were actually available. But even with a good approximation of an arithmetic oracle, that is not practical (the non-temporal first-order formula would be too complex to prove automatically or interactively). And, indeed, the calculus $\mathcal{C}_{\mathrm{DTL}}$ does not work that way.

The (relative) completeness of $\mathcal{C}_{\mathrm{DTL}}$ requires an expressive computation domain and is lost if a simpler domain and less expressive data structures are used. The reason is that in a simpler domain it may not be possible to express the required invariants for all possible while loops.

6 Conclusions and Further Directions

In this paper, we have defined the logic DTL, which stems from a novel combination of dynamic logic and first-order temporal logic. In contrast to [6,10], there is no restriction on the shape of trace formulae. Through this, we have got an expressive logic allowing to describe complex temporal properties of programs. An example proof can be found in Figure 1. Of course, this is a fairly simple program and trace property, but it already requires some proof steps. More elaborate examples (e.g., including proof splits) cannot be given in this paper due to limited space.

One major aim of this work is to express information flow properties in a concurrent setting. In current work in progress, we have sketched an idea how to reason about possible information flows throughout program execution. We still regard only sequential executions of sub-programs (i.e., threads), but execution traces instead of initial and final states. The rationale behind this is that an attacker may be in control of another thread running on the same memory and thus may read variables at any time. For absence of information flow, we show that traces beginning in states which only differ in the values of secret variables are bisimilar in public observations. In earlier work, the information flow policy of non-interference for sequential programs is expressed through self-composition of dynamic logic formulae [12]. This basic idea can be combined with declassification, i.e., the controlled release of information, under temporal constraints, which means to specify *when* information may be released.

State-based dynamic logics, both for deterministic and indeterministic languages, have the well-known property of compositionality. For example, the formulae $[\pi\ \omega]\varphi$ and $[\pi][\omega]\varphi$ are logically equivalent. This is important since program complexity imports much to the overall complexity of a DL formula. This does not apply to our situation as traces may not be decomposed in general. For purposes like loop invariants (see Table 5), however, program decompositions are indispensable. This has lead us to the auxiliary notation $[\![\pi \mid \omega]\!]\varphi$, which talks about all traces beginning in π but extending into ω. Another possibility to make proofs more feasible would be to introduce additional rules for special, commonly used patterns of trace formulae – such as $\Box\Diamond\gamma$ where γ is a state formula – for which we know that decompositions are sound.

The sequent calculus $\mathcal{C}_{\mathrm{DTL}}$ here has been prototypically implemented in the current development version of the interactive KeY prover. Instead of the simple toy language introduced in this paper, the implemented calculus works on actual Java programs. The efforts so far suggest that most program rules can be adapted straight away from the present rules for the $[\cdot]$ modality.

Acknowledgement. The authors would like to thank Mattias Ulbrich, Andreas Wagner, and the anonymous reviewers for their helpful and supporting comments.

References

1. Abadi, M., Manna, Z.: Nonclausal deduction in first-order temporal logic. Journal of the ACM 37(2), 279–317 (Apr 1990)
2. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL semantics for runtime verification. J. Log. Comput 20(3), 651–674 (2010)
3. Beckert, B.: A dynamic logic for Java Card. In: Proceedings, 2nd ECOOP Workshop on Formal Techniques for Java Programs, Cannes, France. pp. 111–119 (2000)
4. Beckert, B., Bruns, D.: Dynamic trace logic: Definition and proofs. Tech. Rep. 2012-10, Karlsruhe Institute of Technology, Department of Computer Science (2012), revised version available at `http://formal.iti.kit.edu/~bruns/papers/trace-tr.pdf`.

5. Beckert, B., Hähnle, R., Schmitt, P.H.: Verification of Object-Oriented Software: The KeY Approach, Lecture Notes in Computer Science, vol. 4334. Springer-Verlag, Berlin (2007)
6. Beckert, B., Schlager, S.: A sequent calculus for first-order dynamic logic with trace modalities. In: Goré, R., Leitsch, A., Nipkow, T. (eds.) Proceedings, International Joint Conference on Automated Reasoning, Siena, Italy. pp. 626–641. LNCS 2083, Springer (2001)
7. Goré, R.: Tableau methods for modal and temporal logics. In: D'Agostino, M., Gabbay, D., Hähnle, R., Posegga, J. (eds.) Handbook of Tableau Methods, pp. 297–396. Kluwer Academic Publishers, Dordrecht (1999)
8. Harel, D.: Dynamic logic. In: Gabbay, D., Guenther, F. (eds.) Handbook of Philosophical Logic, Volume II: Extensions of Classical Logic, pp. 497–604. D. Reidel Publishing Co., Dordrecht (1984)
9. Moszkowski, B.: A temporal logic for multilevel reasoning about hardware. IEEE Computer 18(2) (Feb 1985)
10. Platzer, A.: A temporal dynamic logic for verifying hybrid system invariants. In: Artëmov, S.N., Nerode, A. (eds.) Logical Foundations of Computer Science, 5th International Symposium, LFCS'07, New York, USA, June 4-7, 2007, Proceedings. LNCS, vol. 4514, pp. 457–471. Springer (2007)
11. Reynolds, M., Dixon, C.: Theorem-proving for discrete temporal logic. In: Fisher, M., Gabbay, D., Vila, L. (eds.) Handbook of temporal reasoning in artificial intelligence. Elsevier Science (2005)
12. Scheben, C., Schmitt, P.H.: Verification of information flow properties of Java programs without approximations. In: Formal Verification of Object-Oriented Software, pp. 232–249. LNCS 7421, Springer (2012)
13. Schellhorn, G., Tofan, B., Ernst, G., Reif, W.: Interleaved programs and rely-guarantee reasoning with ITL. In: Combi, C., Leucker, M., Wolter, F. (eds.) Eighteenth International Symposium on Temporal Representation and Reasoning, TIME 2011. pp. 99–106. IEEE (2011)
14. Thums, A., Schellhorn, G., Ortmeier, F., Reif, W.: Interactive verification of statecharts. In: Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (eds.) Integration of Software Specification Techniques for Applications in Engineering. Lecture Notes in Computer Science, vol. 3147, pp. 355–373. Springer (2004)
15. Wolper, P.: The tableau method for temporal logic: An overview. Logique et Analyse 28(110–111), 119–136 (Jun–Sep 1985)