Karlsruhe Reports in Informatics 2012,10
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Dynamic Trace Logic: Definition and Proofs

Bernhard Beckert and Daniel Bruns

2012

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Dynamic Trace Logic: Definition and Proofs*

Bernhard Beckert and Daniel Bruns

Karlsruhe Institute of Technology, Department of Informatics

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 functional verification of concurrent programs and for proving information-flow properties among other applications.

1 Introduction

Overview. Dynamic logic is an established instrument for program verification and for reasoning about the semantics of programs and programming languages. We define an extension of dynamic logic, called Dynamic Trace Logic, which combines the expressiveness of program logics such as DL with that of temporal logic. And we present a sound and relatively complete sequent calculus for proving validity of DTL formulae.

Dynamic logic (DL) [13] is a multi-modal first-order logic. Each legal sequential program fragment π (i.e., a sequence of statements) gives rise to modal operators $[\pi]$ and $\langle\pi\rangle$. The formula $[\pi]\phi$ expresses "in any state in which π terminates, ϕ holds," while the dual $\langle\pi\rangle\phi$ expresses "there is a state in which π terminates and ϕ holds in that one". If programs are deterministic – i.e., there is at most one final state – the modality $\langle\cdot\rangle$ is a variant of $[\cdot]$ which demands termination. Programs in languages like Java are deterministic in the sense that, under some assumptions about the environment (e.g., the presence of unlimited memory), the program represents a function from one system state to another.

Program logics like DL are more expressive than Hoare logics in that programs are part of formulae, and functional properties relating to unbounded data structures can be expressed. In other regards, however, standard dynamic logic lacks expressivity: The semantics of a program is a relation between states; formulae can only describe the input/output behaviour of programs. Standard dynamic logic cannot be used to reason about program behaviour not manifested in

* 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)".

the input/output relation. 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 extends DL with the well-known temporal operators $\Box$ (throughout), $\Diamond$ (eventually), $\bullet$ (weak next), $\circ$ (strong next), $\mathbf{U}$ (until), $\mathbf{W}$ (weak until), and $\mathbf{R}$ (release). And we use trace-based program semantics similar to those of (finite) Linear Temporal Logic (LTL) [14] or Interval Temporal Logic (ITL) [15]. These temporal operators are concerned with *future* states. *Past* operators, such as 'once' or 'since', are not considered here since they do not introduce any additional expressive power (cf. [8]).

In DTL, the formula $[\![\pi]\!]\phi$ expresses that ϕ holds for the (possibly infinite) trace of the program π when started in the current state. For example, the formula

$$[\![\pi]\!]\Box\forall u.\forall v.(X \doteq u \wedge \bullet(X \doteq v) \rightarrow u \leq v)$$

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.

Since programs are included in formulae of DTL, we can have both state and trace formulae in a sequent at the same time – and even formulae expressing how different traces of programs relate to each other. This allows to express information-flow properties by stating that the traces of some program π that result from different secret inputs are sufficiently similar as not to make secret information observable during program execution.

Standard dynamic logic is covered by DTL because the semantics of the standard $[\cdot]$ and $\langle\cdot\rangle$ modalities can be expressed in DTL: The formula $\bullet\mathit{false}$ holds exactly on a trace with only one (remaining) state, thus characterizing termination. We are then able to represent $[\pi]\phi$ by $[\![\pi]\!]\Box(\bullet\mathit{false} \rightarrow \phi)$ and $\langle\pi\rangle\phi$ by $[\![\pi]\!]\Diamond(\bullet\mathit{false} \wedge \phi)$.

Target Programming Language. In the following, we use a simple while language as target programming language without method calls or any feature of object-orientation. However, our language distinguishes between local variables and global variables stored on a heap.

Of course, to be useful in practice, DTL needs to be extended to real-world programming languages. The KeY verification system (co-developed by the authors) is built on a calculus for JavaDL, a dynamic logic for sequential Java [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 [7], we have extended Dynamic Logic with a modality $[\![\cdot]\!]$, where $[\![\pi]\!]\phi$ 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 $[\![\pi]\!]\Box\phi$. That is, in our earlier work, the temporal formula was restricted to the form $\Box\phi$ with ϕ not containing further temporal operators.

Reasoning about temporal properties is traditionally the domain of model checking. 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 linerar-time logics are an article by Wolper [21] and the textbook chapters by Goré [11] and Reynolds and Dixon [17]. The work by Wolper introduces a tableau method for propositional LTL. It is known that, although this logic 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 $\Diamond$). This fixpoint method is strongly related to our approach for handling $\Diamond$ using variants (see rule R29 in Table 5). A similar approach can be found in work by Abadi and Manna [1,2], which is then extended to a first-order version of LTL.

Other related work in the area of program verification w.r.t. temporal specifiations is by Schellhorn et al. [19], who embed programs into ITL formulae. In an earlier work, the authors have also presented a sequent calculus for ITL [20], 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 Sections 2 resp. 3 (including syntax and semantics of the while language that we use as target programming language in this paper). In Section 4, we present our sequent calculus for DTL. Notions of soundness and completeness are defined in Section 5, and we sketch soundness and completeness proofs. The use of our calculus is illustrated with an example in Section 6. Finally, in Section 7, we draw conclusions and discuss further work. The appendix contains a complete proof of soundness and an in-depth account on how to prove completeness of the calculus.

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.[1] Quantifiers can only range over logical variables and not over program variables.

[1] Rigid variables are essential to the expressiveness of the logic. Without them it would be impossible to compare values in different states. E.g., expressing "X has increased by 1" requires to introduce a rigid variable z which in every state evaluates to the pre-state value of X.

Most program logics restrict the syntax such that logical variables may not appear in programs. Our definition is more liberal as expressions in programs may contain

In the basic version of DTL presented 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* $+$, $-$, $*$, $/$ *(integer division),* $\%$ *(integer division remainder). Expressions of type boolean are constructed using the relations* $\doteq$, $>$, $<$ *on integer expressions, the boolean literals true and false, and the logical operators* $\wedge$, $\vee$, $\neg$. *An expression is called a* program expression *if it does not contain any logical variables.*

Programs. Programs are written in a simple while language, with the (mathematical) integers as the only data types. 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 not object-oriented features. As discussed above, all such features can be added, but we keep the programming 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 Section 3, we consider assignments to global variables to be the only program statements that lead to a new observable state. 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 every loop contains an assignment to a global variable. This technical restriction can easily be fulfilled by adding nop-assignments `L=L`.

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

- `v = x;` *where* `v` $\in$ *LVar and* `x` *is a program expression of type integer* (assignment to local variable)*,*
- `G = x;` *where* `G` $\in$ *GVar and* `x` *is a program expression of type integer* (assignment to global variable)*,*
- `if (e) {`π_1`} else {`π_2`}` *where* `e` *is a program expression of type boolean and* π_1 *and* π_2 *are programs* (conditional)*, or*
- `while (e) {`π`}` *where* `e` *is a program expression of type boolean and* π *is a program that contains at least one assignment to a global variable* (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 Section 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

logical variables. As stated in Def. 2, however, logical variables may not occur on the left-hand side of an assignment.

manner. In order to capture the state transitions in between, we use *state updates*. 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 v be a (local or global) program variable, and let e be an expression. Then, $\{v := e\}$ 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 $\mathcal{U}[\![\pi]\!]\phi$ 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, $\mathcal{U}[\![\pi]\!]\phi$ 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 state formulae are also trace formulae.*
1. *All boolean expressions (Def. 1) are state formulae.*
2. *If ϕ and ψ are (state or trace) formulae, then the following are trace formulae: $\Box\phi$ (always), $\bullet\,\phi$ (weak next), $\phi\mathbf{U}\psi$ (until).*
3. *If $\mathcal{U}$ is an update and ϕ a state formula, then $\mathcal{U}\phi$ is a state formula.*
4. *If π is a program and ϕ a trace formula, then $[\![\pi]\!]\phi$ 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:

$$\begin{array}{rclll} \Diamond\phi & := & \neg\Box\neg\phi & \textit{(evntually)}, & \circ\,\phi := \neg\bullet\neg\phi \quad \textit{(strong next)},\\ \phi\mathbf{W}\psi & := & \phi\mathbf{U}\psi \vee \Box\phi & \textit{(weak until)}, & \phi\mathbf{R}\psi := \neg(\neg\phi\mathbf{U}\neg\psi) \quad \textit{(release)},\\ \phi \vee \psi & := & \neg(\neg\phi \wedge \neg\psi), & & \phi \rightarrow \psi := \neg\phi \vee \psi,\\ \exists x.\phi & := & \neg\forall x.\neg\phi. & & \end{array}$$

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).

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

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

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 $\tau\{x \mapsto d\}$.

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:

- $|\tau| \in \mathbb{N} \cup \{\infty\}$ is the *length* of a trace τ. If $\tau = \langle s_0, \dots s_k \rangle$, then $|\tau| = k + 1$.
- $\tau_1 \cdot \tau_2$ is the *concatenation* of traces:
  - If $|\tau_1| = \infty$, then $\tau_1 \cdot \tau_2 = \tau_1$.
  - If $\tau_1 = \langle s_0, \dots, s_k \rangle$ (finite) and $\tau_2 = \langle t_0, \dots \rangle$ (possibly infinite), then $\tau_1 \cdot \tau_2 = \langle s_0, \dots, s_k, t_0, \dots \rangle$.
- $\tau[i, j)$ 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:
  - If $i \geq |\tau|$ or $i \geq j$, then $\tau[i, j) = \tau$
  - If $i < |\tau| < j$, then $\tau[i, j) = \tau[i, |\tau|)$
  - If $\tau = \langle s_0, \dots, s_i, s_{i+1}, \dots, s_{j-1}, s_j, \dots \rangle$, then $\tau[i, j) = \langle s_i, s_{i+1}, \dots, s_{j-1} \rangle$ for $j < \infty$ and $\tau[i, \infty) = \langle s_i, s_{i+1}, \dots \rangle$.
- $\tau[i]$ for $i \in \mathbb{N}$ is the state at position i in τ (with $\tau[i] := \tau[0]$ for $i \geq |\tau|$).

Definition 7 (Semantics of expressions). *Given a state s and a variable assignment β, the valuation $e^{s,\beta}$ of an expression e in a state s is the integer or boolean value resulting from interpreting program variables v by $s(v)$, logical variables x by $\beta(x)$, and using the standard interpretation for all functions and relations.*[2]

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

As mentioned in Section 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 $[\![\pi]\!]\Box\phi$ 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 smallest fixpoint of):*

[2] Values of divisions by zero and its remainders are left *underspecified*. We assume special functions $dbz, mbz : \mathbb{Z} \to \mathbb{Z}$ such that for $b^{s,\beta} = 0$ it is $(a/b)^{s,\beta} := dbz(a)$ and $(a\%b)^{s,\beta} := mbz(a)$.

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

We have now everything needed to define the semantics of DTL formulae in a straightforward way. The valuation of a state formula is given w.r.t. a state s and a variable assignment β; and the valuation of a trace 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 state formulae). *Let s be a state and let β be a variable assignment.*

$$
\begin{array}{lll}
s, \beta \vDash e & \textit{iff} & e^{s,\beta} = \textit{true (in case } e \textit{ is an expression, see Def. 7)} \\
s, \beta \vDash \neg\phi & \textit{iff} & s, \beta \nvDash \phi \\
s, \beta \vDash \phi \wedge \psi & \textit{iff} & s, \beta \vDash \phi \textit{ and } s, \beta \vDash \psi \\
s, \beta \vDash \forall x.\phi & \textit{iff} & \textit{for every } d \in \mathbb{Z}\textit{: } s, \beta\{x \mapsto d\} \vDash \phi \\
s, \beta \vDash [\![\pi]\!]\phi & \textit{iff} & trc(s, \pi), \beta \vDash \phi \qquad \textit{(Def. 10)} \\
s, \beta \vDash \{v := x\}\phi & \textit{iff} & s\{v \mapsto x^s\}, \beta \vDash \phi
\end{array}
$$

A state formula ϕ is valid *if $s, \beta \vDash \phi$ for all s and all β.*

Definition 10 (Semantics of trace formulae). *Let τ be a trace and β a variable assignment.*

$$
\begin{array}{lll}
\tau, \beta \vDash \neg\phi & \textit{iff} & \tau, \beta \nvDash \phi \\
\tau, \beta \vDash \phi \wedge \psi & \textit{iff} & \tau, \beta \vDash \phi \textit{ and } \tau, \beta \vDash \psi \\
\tau, \beta \vDash \forall x.\phi & \textit{iff} & \textit{for every } d \in \mathbb{Z}\textit{: } \tau, \beta\{x \mapsto d\} \vDash \phi \\
\tau, \beta \vDash \Box\phi & \textit{iff} & \tau[i, \infty), \beta \vDash \phi \textit{ for every } i \in [0, |\tau|) \\
\tau, \beta \vDash \phi \mathbf{U} \psi & \textit{iff} & \tau[0, i), \beta \vDash \Box\phi \textit{ and } \tau[i, \infty), \beta \vDash \psi \textit{ for some } i \in [0, |\tau|) \\
\tau, \beta \vDash \bullet\phi & \textit{iff} & \tau[1, \infty), \beta \vDash \phi \textit{ or } |\tau| = 1 \\
\tau, \beta \vDash \gamma & \textit{iff} & \tau[0], \beta \vDash \gamma \textit{ (in case } \gamma \textit{ is a state formula, see Def. 9)}
\end{array}
$$

A trace formula ϕ is valid *if $\tau, \beta \vDash \phi$ 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 Section 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 rules Rules for simplifying formulae of the form $\mathcal{U}[\![\pi]\!]\phi$, where the top-level operator in ϕ is not temporal.

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

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 and/or variant 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).

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, and e is an expression.

Definition 11 (Sequent). *A sequent is a pair of multi-sets of (state) formulae written as* $\gamma_1, \ldots, \gamma_m \vdash \delta_1, \ldots, \delta_n$. *The multi-set* $\{\gamma_1, \ldots, \gamma_m\}$ *of formulae on the left-hand side of the sequent arrow* $\vdash$ *is called the* antecedent*, the set* $\{\delta_1, \ldots, \delta_n\}$ *is called the* succedent *of the sequent. We use capital greek letters to denote subsets of formula, e.g., the sequent notion* $\Gamma, \phi \vdash \psi, \Delta$ *means that formulae* ϕ *and* ψ *occur in the antecedent or succedent and the sets of remaining formulae are* Γ *and* Δ*, respectively.*

A sequent $\Gamma \vdash \Delta$ *is* valid *(in state* s *and under variable assignment* β*) if and only if the formulae* $\bigwedge_{\gamma \in \Gamma} \gamma \rightarrow \bigvee_{\delta \in \Delta} \delta$ *is valid (w.r.t.* s, β*).*

As usual, the sequents above the horizontal line in a schema are its premisses and the single 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 12 (Calculus, derivability). *The* calculus $\mathcal{C}_{\mathrm{DTL}}$ *consists of the rules R1 to R35 shown in Tables 1–7.*

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 are instances of the conclusion of a rule that has no premisses (rules R22, R31, and R34).*

4.1 Classical Logic and Simplification Rules

The rules for quantifiers and propositional operators 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:

$$\frac{\Gamma \vdash \phi, \Delta}{\Gamma, \neg\phi \vdash \Delta}\ \mathsf{R1} \qquad \frac{\Gamma, \phi \vdash \Delta}{\Gamma \vdash \neg\phi, \Delta}\ \mathsf{R2}$$

$$\frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi \wedge \psi \vdash \Delta}\ \mathsf{R3} \qquad \frac{\Gamma \vdash \phi, \Delta \qquad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \wedge \psi, \Delta}\ \mathsf{R4}$$

$$\frac{\Gamma, \phi[x/e], \forall x.\phi \vdash \Delta}{\Gamma, \forall x.\phi \vdash \Delta}\ \mathsf{R5} \qquad \frac{\Gamma \vdash \phi[x/x'], \Delta}{\Gamma \vdash \forall x.\phi, \Delta}\ \mathsf{R6}$$

$$\frac{\Gamma, \mathcal{U}\phi[v\sharp x] \vdash \Delta}{\Gamma, \mathcal{U}\{v := x\}\phi \vdash \Delta}\ \mathsf{R7} \qquad \frac{\Gamma \vdash \mathcal{U}\phi[v\sharp x], \Delta}{\Gamma \vdash \mathcal{U}\{v := x\}\phi, \Delta}\ \mathsf{R8}$$

Table 1. Rules for quantifiers and propositional operators. In rule R5, the substitution needs to be admissible; rule R6 introduces a fresh variable x'. Rules R7 and R8 make use of weak substitution (Def. 14).

Definition 13 (Admissible substitution). *A substitution x/e of a logical variable $x \in V$ with an expression e is* admissible *w.r.t. a formula ϕ if there is no variable y in e such that x is free in ϕ and, after replacing e for some free occurrence of x in ϕ, the occurrence of y in e is* (i) *bound by a quantifier in $\phi[x/e]$ (in case y is a logical variable) or is* (ii) *in the scope of a program modality $[\![\pi]\!]$ that contains an assignment to y (in case y is a program variable).*

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

Definition 14 (Weak substitution). *For a state formula ϕ and an update $\{v := x\}$ define the formula $\phi[v\sharp x]$ according to the following schema:* (i) *if ϕ is an expression, then $\phi[v\sharp x] = \phi[v/x]$,* (ii) *if ϕ begins with an update or a program modality, then $\phi[v\sharp x] = \{v := x\}\phi$,* (iii) *if ϕ is a propositional junction, then the weak substitution is propagated, e.g., $(\phi_1 \wedge \phi_2)[v\sharp x] = \phi_1[v\sharp x] \wedge \phi_2[v\sharp x]$,* (iv) *if ϕ begins with a quantifier, then the weak substitution is propagated (possibly under renaming the bound variable such that it does not occur in x).*

4.2 Simplification Rules

As said above, our calculus contains simplification rules that apply to formulae of the form $\mathcal{U}[\![\pi]\!]\phi$, where the top-level operator in ϕ is not temporal. They are shown in Table 2.

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 $[\![\pi]\!](\Box\phi \wedge \psi)$.

Rule R12 for negated until avoids introducing the dual $\mathbf{R}$ into the sequent. Soundness of R12 follows from the well-known equivalence $\phi\mathbf{R}\psi \leftrightarrow \psi\mathbf{W}(\phi \wedge \psi)$ in LTL and the definitions of $\mathbf{R}$ and $\mathbf{W}$, which carries over to finite traces (e.g., [3]).

$$\frac{\Gamma \vdash \mathcal{U}[\![\pi]\!]\phi,\ \mathcal{U}[\![\pi]\!]\psi, \Delta}{\Gamma \vdash \mathcal{U}[\![\pi]\!](\phi \vee \psi), \Delta}\ \mathsf{R9} \qquad \frac{\Gamma \vdash \mathcal{U}[\![\pi]\!]\phi, \Delta \qquad \Gamma \vdash [\![\pi]\!]\psi, \Delta}{\Gamma \vdash \mathcal{U}[\![\pi]\!](\phi \wedge \psi), \Delta}\ \mathsf{R10}$$

$$\frac{\Gamma \vdash \mathcal{U}[\![\pi]\!]\neg\phi, \Delta}{\Gamma \vdash \neg\mathcal{U}[\![\pi]\!]\phi, \Delta}\ \mathsf{R11} \qquad \frac{\Gamma \vdash \mathcal{U}[\![\pi]\!]\Box\neg\psi,\ \mathcal{U}[\![\pi]\!](\neg\psi \mathbf{U}(\neg\phi \wedge \neg\psi)), \Delta}{\Gamma \vdash \mathcal{U}[\![\pi]\!]\neg(\phi \mathbf{U}\psi), \Delta}\ \mathsf{R12}$$

$$\frac{\Gamma \vdash \mathcal{U}[\![\pi]\!]\neg\phi, \Delta}{\Gamma, \mathcal{U}[\![\pi]\!]\phi \vdash \Delta}\ \mathsf{R13} \qquad \frac{\Gamma \vdash \mathcal{U}[\![\pi]\!]\phi, \Delta}{\Gamma \vdash \mathcal{U}[\![\pi]\!]\neg\neg\phi, \Delta}\ \mathsf{R14}$$

$$\frac{\Gamma \vdash \mathcal{U}[\![\pi]\!]\circ\neg\phi, \Delta}{\Gamma \vdash \mathcal{U}[\![\pi]\!]\neg\bullet\phi, \Delta}\ \mathsf{R15} \qquad \frac{\Gamma \vdash \mathcal{U}\gamma, \Delta}{\Gamma \vdash \mathcal{U}[\![\pi]\!]\gamma, \Delta}\ \mathsf{R16}$$

$$\frac{\Gamma \vdash \mathcal{U}[\![\pi]\!]\phi[x/x'], \Delta}{\Gamma \vdash \mathcal{U}[\![\pi]\!]\forall x.\phi, \Delta}\ \mathsf{R17} \qquad \frac{\Gamma \vdash \mathcal{U}[\![\pi]\!]\phi[x/e], \mathcal{U}[\![\pi]\!]\exists x.\phi, \Delta}{\Gamma \vdash \mathcal{U}[\![\pi]\!]\exists x.\phi, \Delta}\ \mathsf{R18}$$

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

$$\frac{\Gamma \vdash \mathcal{U}([\![\pi]\!]\circ(\phi \mathbf{U}\psi) \wedge [\![\pi]\!]\phi),\ \mathcal{U}[\![\pi]\!]\psi, \Delta}{\Gamma \vdash \mathcal{U}[\![\pi]\!]\phi \mathbf{U}\psi, \Delta}\ \mathsf{R19} \qquad \frac{\Gamma \vdash \mathcal{U}[\![\pi]\!]\bullet\Box\phi, \Delta \qquad \Gamma \vdash \mathcal{U}[\![\pi]\!]\phi, \Delta}{\Gamma \vdash \mathcal{U}[\![\pi]\!]\Box\phi, \Delta}\ \mathsf{R20}$$

$$\frac{\Gamma \vdash \mathcal{U}[\![\pi]\!]\circ\Diamond\phi,\ \mathcal{U}[\![\pi]\!]\phi, \Delta}{\Gamma \vdash \mathcal{U}[\![\pi]\!]\Diamond\phi, \Delta}\ \mathsf{R21} \qquad \frac{}{\Gamma \vdash \mathcal{U}[\![]\!]\bullet\phi, \Delta}\ \mathsf{R22} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \mathcal{U}[\![]\!]\circ\phi, \Delta}\ \mathsf{R23}$$

Table 3. Rules for handling temporal operators.

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.

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.

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 R25, 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.

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 R26. 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 than 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

$$\frac{\Gamma \vdash \mathcal{U}\{v := a\}[\![\omega]\!]\phi, \Delta}{\Gamma \vdash \mathcal{U}[\![\texttt{v = a; } \omega]\!]\phi, \Delta}\ \mathsf{R24} \qquad \frac{\Gamma \vdash \mathcal{U}\{G := a\}[\![\omega]\!]\phi, \Delta}{\Gamma \vdash \mathcal{U}[\![\texttt{G = a; } \omega]\!]\circledcirc\phi, \Delta}\ \mathsf{R25}$$

$$\frac{\Gamma, \mathcal{U}b \vdash \mathcal{U}[\![\pi_1\ \omega]\!]\phi, \Delta \qquad \Gamma, \mathcal{U}\neg b \vdash \mathcal{U}[\![\pi_2\ \omega]\!]\phi, \Delta}{\Gamma \vdash \mathcal{U}[\![\texttt{if (b) } \pi_1 \texttt{ else } \pi_2\ \omega]\!]\phi, \Delta}\ \mathsf{R26}$$

$$\frac{\Gamma, \mathcal{U}b \vdash \mathcal{U}[\![\pi \texttt{ while (b) } \pi\ \omega]\!]\phi, \Delta \qquad \Gamma, \mathcal{U}\neg b \vdash \mathcal{U}[\![\omega]\!]\phi, \Delta}{\Gamma \vdash \mathcal{U}[\![\texttt{while (b) } \pi\ \omega]\!]\phi, \Delta}\ \mathsf{R27}$$

Table 4. Program rules. The symbol ◎ stands for either weak or strong next.

iterations may not be known in advance, or the loop may not even terminate. In those cases, we need loop rules using invariants.

Invariant rules are an established technique for handling loops in calculi for program logics. Indeed, the invariant rule R28 for a box operator after the program modality is very similar to the invariant rule in standard dynamic logic. It has three premisses: The first states that the invariant *Inv*, which is an arbitrary DTL formula, holds in the initial state. The second premiss states that (a) *Inv* is preserved through execution of the loop body (i.e., it actually is an invariant) and $\Box\phi$ holds throughout the trace given by $[\![\pi]\!]$. Note that *Inv* does not need to hold throughout the loop body, but only at the end. This is expressed by the condition $\bullet\mathit{false}$, which holds if and only if the remaining trace contains exactly one state. The third premiss states: After the loop has terminated, i.e., in a state where the invariant holds and the loop condition does not hold, the rest of the program has the original temporal property. As an invariant abstracts from concrete loop iterations, the context Γ, Δ must be discarded in the second and third premiss.

$$\frac{\Gamma \vdash \mathcal{U}\mathit{Inv}, \Delta \qquad \mathit{Inv},\ b \vdash [\![\pi]\!]\Box(\phi \wedge (\bullet\mathit{false} \rightarrow \mathit{Inv})) \qquad \mathit{Inv},\ \neg b \vdash [\![\omega]\!]\Box\phi}{\Gamma \vdash \mathcal{U}[\![\texttt{while}(b)\ \pi\ \omega]\!]\Box\phi, \Delta}\ \mathsf{R28}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathcal{U}\mathit{Inv}, \Delta \qquad V(x) \vdash x \geq 0 \qquad V(x), x > 0 \vdash b \\ \mathit{Inv}, V(x), x > 0, b \vdash [\![\pi]\!]\Diamond(\bullet\mathit{false} \wedge \mathit{Inv} \wedge (V(x') \rightarrow x' < x)) \\ \mathit{Inv}, V(x), x \doteq 0 \vdash [\![\texttt{while}(b)\ \pi\ \omega]\!]\Diamond\phi\end{array}}{\Gamma \vdash \mathcal{U}[\![\texttt{while}(b)\ \pi\ \omega]\!]\Diamond\phi, \Delta}\ \mathsf{R29}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathcal{U}\mathit{Inv}, \Delta \qquad V(x) \vdash x \geq 0 \qquad V(x), x > 0 \vdash b \\ \mathit{Inv}, V(x), x > 0, b \vdash [\![\pi]\!](\Box\phi \wedge \Diamond(\bullet\mathit{false} \wedge \mathit{Inv} \wedge (V(x') \rightarrow x' < x))) \\ \mathit{Inv}, V(x), x \doteq 0 \vdash [\![\texttt{while}(b)\ \pi\ \omega]\!]\phi\mathbf{U}\psi\end{array}}{\Gamma \vdash \mathcal{U}[\![\texttt{while}(b)\ \pi\ \omega]\!]\phi\mathbf{U}\psi, \Delta}\ \mathsf{R30}$$

Table 5. Invariant rules.

Invariant rules for other temporal operators are more tricky. Simply changing $\Box$ to $\Diamond$ in the above rule R28 is not sound. Instead, for handling $\Diamond$ and $\mathbf{U}$, we introduce invariant rules R29 and R30 with five premisses each. The central idea for handling $\Diamond\phi$ is that we first skip over a (finite) number of loop iterations during which ϕ does not become true, then unwind the loop for a small number of times (preferably once), and show that ϕ becomes true during these iterations. As

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

$$\text{if } \bigwedge \Gamma_1 \to \bigwedge \Gamma_1' \text{ is a valid non-temporal formula: } \frac{\Gamma_1', \Gamma_2 \vdash \Delta}{\Gamma_1, \Gamma_2 \vdash \Delta} \ \mathsf{R32}$$

$$\frac{\Gamma \vdash \phi(0), \Delta \qquad \Gamma, \phi(n) \vdash \phi(n+1), \Delta}{\Gamma \vdash \forall n.\phi(n), \Delta} \ \mathsf{R33}$$

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

the number of iterations that we skip over must shown to be finite, we introduce a *variant* V. It does not serve as a termination witness here, but as a witness for progress.

A variant is a DTL formula with a free logical variable x. The intuition of a variant V is that the integer values of x for which V is true (typically only one in every state) decreases with each loop iteration but cannot become negative. Positive values imply that the loop is still executed at least once (third premiss). The statement that the variant decreases is included in the forth premiss of R29: For any x' (another free variable) with $V(x')$ in the post-state, it holds that x' is strictly less than x. The condition that x must be non-negative is captured in the second premiss. In contrast to rule R28, the fifth premiss (use case) still contains the original temporal operator and the `while` statement – but under the assumption that $V(x)$ and $x \doteq 0$ are true.

In practice, most variants take the form $x \doteq e$, where the integer expression e decreases with each loop iteration. In general, however, it is not always possible to fully encode the behaviour of a loop in an expression but only in a complex formula using quantifiers.

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. Rule R32 is an alternative formalization of the oracle that is often more useful.

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 rules R31 and R32 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 programs. And using powerful SMT solvers, this can be done fully automatically in many cases.

$$\frac{}{\Gamma, \phi \vdash \phi, \Delta}\ \mathsf{R34} \qquad \frac{\Gamma, \phi \vdash \Delta \qquad \Gamma \vdash \phi, \Delta}{\Gamma \vdash \Delta}\ \mathsf{R35}$$

Table 7. The closure and the cut rule.

Typically, an approximation of the computation domain oracle contains a rule for structural induction. In the case of arithmetic, that is rule R33. This rule, however, not only applies to non-temporal formulae but also to DTL formulae containing programs.

4.6 Other Rules

The remaining rules, which are shown in Table 7, are the cut rule R35 (with an arbitrary cut formula ϕ) and the closure rule R34 that closes a proof branches.

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 R35, 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., $\bigwedge \Gamma \rightarrow \bigvee \Delta$ 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. This is shown in Appendix 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 one of the oracle rules for arithmetic, R31 and R32. 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 $\vdash \phi$ 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 [13] to prove relative completeness of his sequent calculus for first-order DL. An extensive proof sketch can be found in Appendix B. The following lemma is central to the completeness proof.

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:*

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

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 $\phi_{\psi,s,\pi,n}$ encoding that $\tau[n,\infty) \vDash \psi$, where $\tau = trc(s,\pi)$.

Note that Lemma 1 states a property of the logic DTL that is independent of any calculus.

Lemma 1 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 resp. variants for all possible while loops.

6 Example

Consider the program

$$\texttt{while (true) } \pi \quad \text{with} \quad \pi := \texttt{X = X - X/2;}$$

(where `X` is a global variable). Remember that the slash symbol is interpreted as integer division without remainder, i.e., `X` is assigned $X - \lfloor X/2 \rfloor = \lceil X/2 \rceil$ on each iteration. This program obviously does not terminate for any value of `X`. However, for positive values of X, any execution trace eventually stabilizes in a state where $X \doteq 1$. This property cannot be expressed using standard dynamic logics. In our logic DTL, it can be expressed as $[\![\pi]\!]\Diamond\Box(X \doteq 1)$.

Figure 1 shows a complete proof tree (in two parts), which does not involve any loop unwinding. The proof starts in the lower part. The first rule application is the invariant rule for diamond, R29, where the invariant is *true* and the variant $V(y)$ is $y \geq 0 \wedge y \doteq X - 1$. The two branches on the left can be closed immediately. In the second branch, we have to prove the step case: If the variant is strictly greater than zero, then it is greater or equal to zero after a loop iteration. Here, the box operator is unwound (rule R20); the resulting conjunction is then split up. The resulting right branch can be closed easily by falsifying that the trace is empty. In the left branch, we first apply rule R25 and then again rule R20. While the "future" branch closes instantaneously, in the "present" branch, we have to show that there is a decrease of the variant.

The third branch (use case) of the initial rule application is shown in the upper part of the figure. The variant now is exactly zero. We apply rule R21 and hide the "future" part. Now we can apply the invariant rule to a box formula.

The first and third premiss close in a few steps. In the second premiss (invariant preservation), we first apply rule R20 again. The resulting right branch ("present") closes similar as shown before. In the left branch ("future"), the loop body is symbolically executed (rule R25) and then rule R20 is applied once again. All resulting branches close in a few steps.

7 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. Through this, we got an expressive logic allowing to describe complex temporal properties of programs.

The sequent calculus $\mathcal{C}_{\mathrm{DTL}}$ here has been prototypically implemented in the current development version of the 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 must program rules can be adapted straight away from the present rules for the $[\cdot]$ and $\langle\cdot\rangle$ modalities since they are non-stepping in the semantics presented in this paper. The calculus for JavaDL has been proven sound and complete [4]; this provides us some confidence that also a trace-based calculus for Java will be sound and complete. We will develop notations for the temporal properties in the Java Modeling Language (JML) which is the main specification interface of the KeY system.

As an immediate follow-up work, we will investigate heuristics for proof strategies. In standard dynamic logic calculi (for deterministic languages), program transformation rules usually have a high priority since most of them do not split the proof while there are few rules which rewrite sub-formulae below modalities. This is different for our calculus. Therefore, it becomes an issue of proof complexity whether first to symbolically execute the program or to rewrite the formula below the program modality.

One major aim of this work is to express information flow properties. It seems reasonable to incorporate specification means such as the "hide until" operator $\mathcal{H}_{I,O}$ of the SecLTL logic [9].

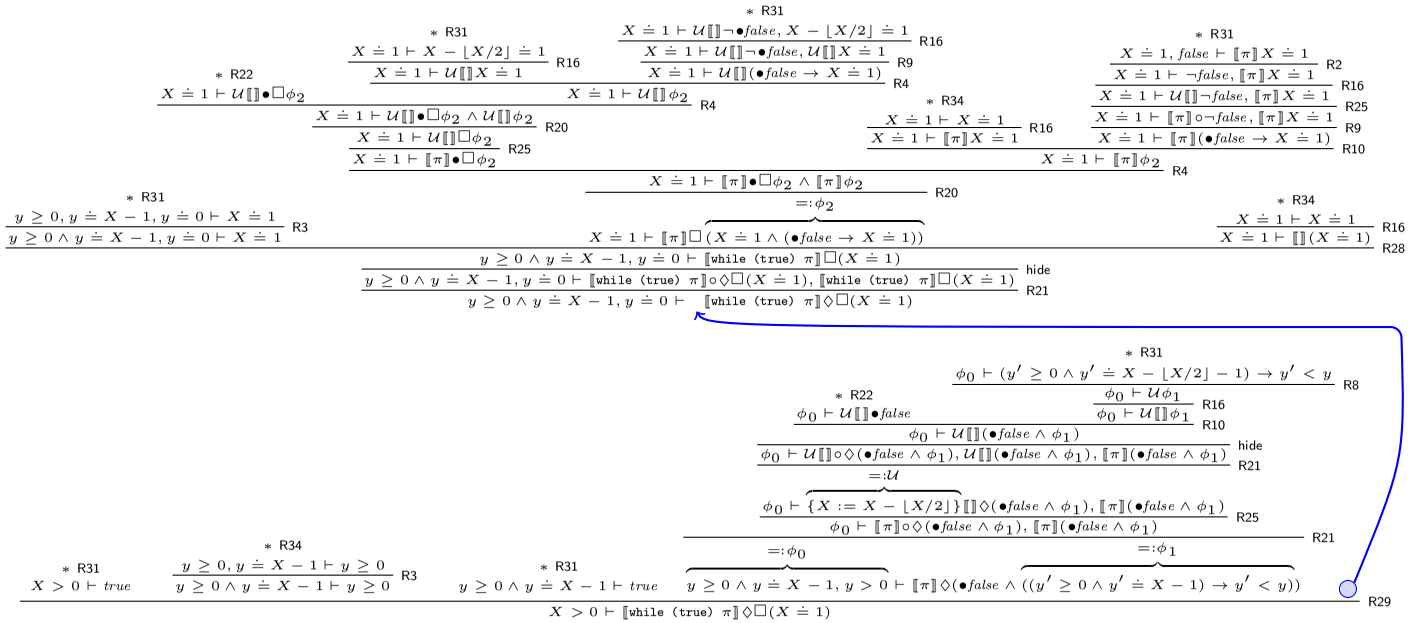

Fig. 1. Example proof for the formula $[\![\texttt{while (true) } \pi]\!]\Diamond\Box(X \doteq 1)$. From the root (bottom), application of rule R29 yields four premisses, of which the forth is shown above.

A Soundness Proofs

This appendix contains lemmas and respective proofs from which Theorem 1 follows as a corollary. Since most proof techniques reappear in the proof to each lemma, we will gradually reduce the proofs' details.

Lemma 2 (Soundness of propositional, quantifier, and update rules). *Rules R1–R8, R34, and R35 are sound.*

Proofs for rules R1–R6, R34, and R35 (propositional and quantifier rules) can be found many logics text books (see, e.g., [10]). Note for rule R6 that free logical variables implicitly are universally quantified. Proofs for rules R7 and R8 (update rules) can be found in [6], where a sequent calculus for a simple dynamic logic is presented.

Definition 15. *For a state s and an update $\mathcal{U}$, we introduce a state $s^{\mathcal{U}}$ which only differs from s in that it is updated for every update in $\mathcal{U}$. More formally, $s^{\mathcal{U}} = s_k$ with $s_i := s_{i-1}\{v_i \mapsto x_i^{s_{i-1}}\}$ and $s_0 := s$ for $\mathcal{U} = \{v_1 := x_1\}\{v_2 := x_2\}\cdots\{v_k := x_k\}$.*

Lemma 3 (Ommission of environments). *A rule of the shape*

$$\frac{\Gamma, \Phi_1 \vdash \Psi_1, \Delta \quad \cdots \quad \Gamma, \Phi_k \vdash \Psi_k, \Delta}{\Gamma, \Phi \vdash \Psi, \Delta} \tag{1}$$

is sound if and only if the following rule is sound:

$$\frac{\Phi_1 \vdash \Psi_1 \quad \cdots \quad \Phi_k \vdash \Psi_k}{\Phi \vdash \Psi} \tag{2}$$

Proof. The one proof direction, from sequent (1) to (2), is trivial since it is a weakening. For the other direction, assume the sequents $\Gamma, \Phi_i \vdash \Psi_i, \Delta$ valid for all i. This means that $\Gamma \wedge \Phi_i \rightarrow \Psi_i \vee \Delta$ is valid. Assume $\Gamma \rightarrow \Delta$ invalid. (Otherwise the conclusion would be trivially valid.) This means that $\Phi_i \vdash \Psi_i$ is valid and from (2) it follows that $\Phi \vdash \Psi$ is valid. Since Γ is invalid and Δ is valid, the conclusion of (1) is a weakening of that sequent. △

Since in all rules—except invariant rules—the contexts Γ and Δ are preserved in the conclusion, WLOG, we assume them to be empty.

Lemma 4 (Soundness of rules for promotion/demotion of propositional operators). *Rules R9–R11 are sound.*

Proof. We show the proof for R9 ("pull out or"); the remaining proofs can be obtained in a similar way. Assume the following sequent valid: $\vdash \mathcal{U}[\![\pi]\!]\phi, \mathcal{U}[\![\pi]\!]\psi,$. By Def. 11, for any state s and valuation β; $s, \beta \vDash \mathcal{U}[\![\pi]\!]\phi$ or $s, \beta \vDash \mathcal{U}[\![\pi]\!]\psi$ holds. Then the above validity assumption is equivalent to $s^{\mathcal{U}}, \beta \vDash [\![\pi]\!]\phi$ or $s^{\mathcal{U}}, \beta \vDash [\![\pi]\!]\psi$ according to Def. 9; and again to $trc(s^{\mathcal{U}}, \pi), \beta \vDash \phi$ or $trc(s^{\mathcal{U}}, \pi), \beta \vDash \psi$. According to Def. 10, this is again equivalent to $trc(s^{\mathcal{U}}, \pi), \beta \vDash \phi \vee \psi$. Applying the above definitions in the opposite direction yields $s, \beta \vDash \mathcal{U}[\![\pi]\!](\pi \vee \psi)$. △

Lemma 5. *Rule R13 is sound.*

Proof. Assume the following sequent valid: $\vdash \mathcal{U}[\![\pi]\!]\neg\phi$. By definition, for any state s and valuation β: $trc(s^{\mathcal{U}}, \pi), \beta \not\models \phi$, which is equivalent to $s, \beta \not\models \mathcal{U}[\![\pi]\!]\phi$. Through the dual nature of sequents (Def. 11), both sequents $\vdash \neg\mathcal{U}[\![\pi]\!]\phi$ and $\mathcal{U}[\![\pi]\!]\phi \vdash$ are valid. △

Lemma 6. *Rule R15 ("negation next") is sound.*

Proof. Immediately follows from the dual definition.

Lemma 7. *Rule R12 ("negation until") is sound.*

Proof. Assume the following sequent valid:

$$\vdash \mathcal{U}[\![\pi]\!]\Box\neg\psi, \mathcal{U}[\![\pi]\!](\neg\psi \mathbf{U}(\neg\phi \wedge \neg\psi))$$

Let $\tau := trc(s^{\mathcal{U}}, \pi)$; at least one the following relations hold: $\tau, \beta \vDash \Box\neg\psi$ or $\tau, \beta \vDash \neg\psi \mathbf{U}(\neg\phi \wedge \neg\psi)$. We make the following case distinction: (i) Assume $\tau \vDash \neg\psi \mathbf{U}(\neg\phi \wedge \neg\psi)$. By the definition, there are subtraces τ' and τ'' with $\tau = \tau' \cdot \tau''$ such that $\tau' \vDash \Box\neg\psi$, $\tau'' \vDash \neg\phi$, and $\tau'' \vDash \neg\psi$. Now assume that $\tau \vDash \phi \mathbf{U} \psi$; obviously, this can be contradicted for both subtraces. (ii) Assume $\tau \vDash \Box\neg\psi$; it immediately follows that there is no subtrace τ''' of τ such that $\tau''' \vDash \psi$. △

Lemma 8. *Rule R14 ("double negation") is sound.*

Proof. Immediately follows from Def. 10.

Lemma 9. *Rule R16 ("apply update") is sound.*

Proof. Assume the sequent $\vdash \mathcal{U}\gamma$ valid where γ is a state formula. Thus for every s, β; it holds $s^{\mathcal{U}}, \beta \vDash \gamma$. According to Def. 10, it also holds $\tau, \beta \vDash \gamma$ for every τ with $\tau[0] = s^{\mathcal{U}}$. Since $trc(s', \pi)[0] = s'$ for any state s' and program π, it follows $trc(s^{\mathcal{U}}, \pi), \beta \vDash \gamma$ and $s, \beta \vDash \mathcal{U}[\![\pi]\!]\gamma$ for every program π. △

Lemma 10 (Soundness of rules for quantifiers in trace formulae). *Rules R17 ("forall trace") and R18 ("exists trace") are sound.*

Proof. Similar to the proofs above, it can be shown that for $\exists\forall \in \{\forall, \exists\}$, the formulae $\mathcal{U}[\![\pi]\!]\exists\forall x.\phi$ and $\exists\forall x.\mathcal{U}[\![\pi]\!]\phi$ are equivalent.[3] Since the valuation of x as a logical variable only depends on β and not on the state, rules R17 and R18 are sound if and only if the corresponding rules of pure first-order logic are sound. Note that for the formula $\phi[x/x']$ with a free variable x' to be valid, it means that $\tau, \beta \vDash \phi[x/x']$ for any valuation β, thus having an implicit universal quantification on the semantical level. △

[3] At least under the condition that x does not occur syntactically in $\mathcal{U}$, which can be assumed without loss of generality. (As a logical variable it does not occur in π by definition.)

Lemma 11 (Soundness of unwinding rules). *Rules R19 ("unwind until"), R20 ("unwind box"), and R21 ("unwind diamond") are sound.*

Proof. We show the proof for R19; the other ones follow a similar (and simpler) shape. Assume the following sequent to be valid: $\vdash \mathcal{U}([\![\pi]\!]\circ(\phi \mathbf{U} \psi) \wedge [\![\pi]\!]\phi), \mathcal{U}[\![\pi]\!]\psi$. For any state s and $\tau := trc(s^{\mathcal{U}}, \pi)$ at least one of $\tau \vDash \circ(\phi \mathbf{U} \psi) \wedge \phi$ or $\tau \vDash \psi$ holds. From the second formula, it would immediately follow that $\tau \vDash \phi \mathbf{U} \psi$, so assume it invalid. From the other formula, it follows that both $\tau[1, \infty) \vDash \phi \mathbf{U} \psi$ and $\tau[0] \vDash \phi$. The state formula ϕ can be lifted to a trace formula $\Box\phi$ over a single-state trace: $\tau[0, 1) \vDash \Box\phi$. This matches the definition of semantics of the $\mathbf{U}$ operator and $\tau \vDash \phi \mathbf{U} \psi$ follows. △

Lemma 12. *Rules R22 ("empty trace weak next") and R23 ("empty trace strong next") are sound.*

Proof. The proof to R23 is trivial since the conclusion is a weakening of the premiss. For R22, we have to show that $\mathcal{U}[\![]\!]\bullet\phi$ is valid, which follows immediately from Def. 8. △

Lemma 13. *Rule R24 ("local assignment") is sound.*

Proof. Assume sequent $\vdash \mathcal{U}\{v := a\}[\![\omega]\!]\phi$ valid. I.e., $trc(s^{\mathcal{U}\{v:=a\}}, \omega) \vDash \phi$ for every state s. It is $s^{\mathcal{U}\{v:=a\}} = s^{\mathcal{U}} \left\{ v \mapsto a^{s^{\mathcal{U}}} \right\}$ and thus $trc(s^{\mathcal{U}}, \texttt{v = a; } \omega) \vDash \phi$ according to Def. 8. It follows $s \vDash \mathcal{U}[\![\texttt{v = a; } \omega]\!]\phi$. △

Lemma 14. *Rule R25 ("global assignment") is sound.*

Proof. We show the proof for the case with "weak next". Similar to the proof to Lemma 13 above, we assume $trc(s^{\mathcal{U}} \left\{ G \mapsto a^{s^{\mathcal{U}}} \right\}, \omega) \vDash \phi$ for any s. Then, for any state s', in particular for s, the following holds according to Def. 10:

$$\langle s' \rangle \cdot trc(s^{\mathcal{U}} \left\{ G \mapsto a^{s^{\mathcal{U}}} \right\}, \omega) \vDash \bullet\phi$$

It follows $trc(s^{\mathcal{U}}, \texttt{G = a; } \omega) \vDash \bullet\phi$ and finally $s \vDash \mathcal{U}[\![\texttt{G = a; } \omega]\!]\bullet\phi$. △

Lemma 15. *Rules R26 ("if-then-else") and R27 ("unwind loop") are sound.*

Proof. For R26, assume both sequents $\mathcal{U}b \vdash \mathcal{U}[\![\pi_1\ \omega]\!]\phi$ and $\mathcal{U}\neg b \vdash \mathcal{U}[\![\pi_2\ \omega]\!]\phi$ valid. We do a case-distinction (w.r.t. s) on whether $s^{\mathcal{U}} \vDash b$ holds. Either case amounts to a case in Def. 8 (it is $trc(s^{\mathcal{U}}, \pi_i\ \omega) \vDash \phi$ for either $i \in \{1, 2\}$, respectively), which collectively amounts to $trc(s^{\mathcal{U}}, \texttt{if (b) \{}\pi_1\texttt{\} else \{}\pi_2\texttt{\} } \omega) \vDash \phi$. The proof for R27 follows a similar path. △

Lemma 16 (Soundness of invariant rule for $\Box$). *Rule R28 is sound.*

Proof. Assume the following sequents to be valid: (i) $\vdash \mathcal{U}Inv$, (ii) $Inv, b \vdash [\![\pi]\!]\Box(\phi \wedge (\bullet false \rightarrow Inv))$, (iii) $Inv, \neg b \vdash [\![\omega]\!]\Box\phi$. What is to be shown is that

the sequent $\vdash \mathcal{U}[\![\texttt{while (b) \{}\pi\texttt{\}}\ \omega]\!]\Box\phi$ is valid, i.e., it holds in any state. Let us fix some state s.

1. Assume that the loop executed in state $s^{\mathcal{U}}$ does not terminate. This means the trace of the complete program is equal to an infinite concatenation of the traces yielded by the loop body π. Let the states in which the loop condition is evaluated be denoted by s_i for $i \in \mathbb{N}$, i.e., $s_0 = s^{\mathcal{U}}$ and s_{i+1} is the last state in $trc(s_i, \pi)$ (if such exists). It remains to show $s_i \vDash [\![\pi]\!]\Box\phi$. From premiss (ii) we get that in every state in which Inv and b hold, also $[\![\pi]\!]\Box\phi$ holds. Obviously, $s_i \vDash b$ (otherwise the loop would terminate). From the validity of (i) follows that $s_0 \vDash Inv$ and from (ii) follows that if $s_i \vDash Inv$ then $s_{i+1} \vDash Inv$ since the formula $\bullet\mathit{false}$ is true exactly in the final state of a trace. By induction over i, this closes the case where the loop does not terminate.

2. Let us now assume that the loop takes exactly $n \in \mathbb{N}$ iterations. Let $s_0, \ldots, s_n$ be as above. The proof follows an induction over n.

IH If the loop executed in a state s_i with $s_i \vDash Inv$ takes at most n iterations, then $s_i \vDash [\![\texttt{while (b) \{}\pi\texttt{\}}\ \omega]\!]\Box\phi$.

IA $n = 0$, which means that $s_i \vDash \neg b$ because otherwise there would be another loop iteration. The trace of the complete program therefore is equal to the trace of ω when started in s_i and it remains to show $s_i \vDash [\![\omega]\!]\phi$, which follows from premiss (iii).

IS $n > 0$, which means that $s_i \vDash b$. As we have shown above, for the successor state s_{i+1} of s_i, $s_{i+1} \vDash Inv$ holds and from the induction hypothesis we get $s_{i+1} \vDash [\![\texttt{while (b) \{}\pi\texttt{\}}\ \omega]\!]\Box\phi$. By the definition of successors it follows $s_i \vDash [\![\pi\ \texttt{while (b) \{}\pi\texttt{\}}\ \omega]\!]\Box\phi$. Since the loop condition holds in s_i and we know $s_i \vDash [\![\pi]\!]\Box\phi$ from premiss (ii), this is equivalent to $s_i \vDash [\![\texttt{while (b) \{}\pi\texttt{\}}\ \omega]\!]\Box\phi$.

From premiss (i) follows that the induction hypothesis holds for the initial state $s^{\mathcal{U}}$ in particular. △

Lemma 17 (Soundness of invariant rules for ◇ and U). *Rules R29 and R30 are sound.*

Proof. We only show R30 as R29 is just a special case where $\phi = \mathit{true}$. Assume the following sequents to be valid:

(i) $\vdash \mathcal{U}Inv$,
(ii) $V(x) \vdash x \geq 0$,
(iii) $V(x), x > 0 \vdash b$,
(iv) $Inv, V(x), x > 0, b \vdash [\![\pi]\!](\Box\phi \wedge \Diamond(\bullet \mathit{false} \wedge Inv \wedge (V(x') \rightarrow x' < x)))$,
(v) $Inv, V(x), x \doteq 0 \vdash [\![\texttt{while (b) \{}\pi\texttt{\}}\ \omega]\!]\phi \mathbf{U} \psi$.

What is to be shown is that the sequent $\vdash \mathcal{U}[\![\texttt{while (b) \{}\pi\texttt{\}}\ \omega]\!]\phi\mathbf{U}\psi$ is valid, i.e., it holds in any state. Let us fix some state s. As above, let s_i be defined as $s_0 = s^{\mathcal{U}}$ and s_{i+1} as the final state (if it exists) in $trc(s_i, \pi)$ for $0 \leq i < n$ where $n \in \mathbb{N} \cup \{\infty\}$ is the exact number of loop iterations. We will call s_{i+1} the *successor* of s_i if it exists. Remember that we do not require the loop to

terminate, the rule only states that there is a finite prefix of the trace such that ψ holds eventually on it (and ϕ holds until then).

Auxiliary conjecture 1: *For every s_i with $s_i \vDash Inv \wedge V(x) \wedge x > 0$, there exists a successor state s_{i+1} with $s_{i+1} \vDash Inv$.*
Proof: As above in the proof to Lemma 16, we apply induction over i: The base case immediately follows from premiss (i). In the step case, we get $s_i \vDash b$ from premiss (iii) and (iv) yields both the existence of s_{i+1} and $s_{i+1} \vDash Inv$.

Auxiliary conjecture 2: *There is a state s_k with $0 \leq k \leq n$ and $s_k \vDash Inv \wedge V(x) \wedge x \doteq 0$.*
Proof: Define a function $\chi_V : [0, n] \to \mathbb{Z}$ with $\chi_V(j) = \min\{z \in \mathbb{Z} \mid s_j \vDash V(x) \text{ and } x^{s_j} = z\}$. Premiss (ii) yields that $\chi_V(j) \geq 0$. Apply induction over k. In the base case, $s_0 \vDash [\![\texttt{while (b) \{}\pi\texttt{\}}\ \omega]\!]\phi \mathbf{U} \psi$ follows directly from premiss (v). In the step case, $k > 0$, assume $\chi_V(k) > 0$ (otherwise the proof would conclude with s_k), and thus $s_k \vDash b$. Premiss (iv) yields both the existence of a successor state s_{k+1} (since π terminates when executed in s_k) and $\chi_V(k+1) < \chi_V(k)$. From conjecture 1, we draw that $s_{k+1} \vDash Inv$. If $\chi_V(k+1) > 0$, we perform another induction step. By definition of χ_V, there is no infinite decreasing chain. The case of $\chi_V(k+1) < 0$ is contradicted by premiss (ii), therefore, we conclude $\chi_V(k+1) = 0$.

Auxiliary conjecture 3: *For every $j \in [0, k)$, it is $s_j \vDash [\![\pi]\!]\Box\phi$*
Proof: By construction above, it is $\chi_V(j) > 0$. Also since conjecture 1 yields $s_j \vDash Inv$, the conjecture can be concluded from premiss (iv).

As a corollary from conjecture 2, we obtain $s_k \vDash [\![\texttt{while (b) \{}\pi\texttt{\}}\ \omega]\!]\phi \mathbf{U} \psi$. By construction and since $s_i \vDash b$ for all i, it is

$$trc(s_0, \texttt{while (b) \{}\pi\texttt{\}}) = trc(s_0, \pi) \cdot \ldots \cdot trc(s_{k-1}, \pi) \cdot trc(s_k, \texttt{while (b) \{}\pi\texttt{\}})$$

Since we get $s_0 \vDash [\![\pi \ \cdots \ \pi]\!]\Box\phi$ from conjecture 3, we conclude that $s_0 \vDash [\![\texttt{while (b) \{}\pi\texttt{\}}\ \omega]\!]\phi \mathbf{U} \psi$. △

Lemma 18 (Soundness of arithmetic rules). *Rules R31, R32, and R33 are sound.*

Soundness of R31 and R32 immediately follows from the side-conditions on validity. Rule R33 follows from the induction principle on natural numbers.

B Completeness Proof

In this section, we are about to prove Theorem 2, i.e., that the calculus presented in Sect. 4 is relatively complete. It follows from Lemma 1, which states that any DTL formula can be encoded in first-order logic with arithmetic, and Lemma 19, which states that the calculus entails a complete calculus for first-order logic.

Lemma 19 (First-order completeness). *The rules R1 to R6, R34 and R35 form a complete calculus for first-order predicate logic.*

Proofs of this kind can be found in standard textbooks on first-order logic calculi. In particular, the calculus contains rules for both kinds (left-hand side/ right-hand side) of negation (R1 and R2), an α rule (R3), a β rule (R4), a γ rule (R5), and a δ rule (R6). Together with rules R31 and R33, it is powerful enough to handle arithmetic.

The remainder of this section is laying the foundations for a proof of Lemma 1. By structural induction, we show how a DTL formula can be encoded in first-order logic. For a DTL formula ϕ, we give an equivalent FOL formula $\mathcal{F}(\phi)$.

B.1 Expressibility of Programs

As a first step, we transform programs to formulae according to a single-static-assignment schema and introduce fresh logical variables for every program variable and state on the program trace. The completeness proof for the dynamic logic of [12], however, contains an error as the translation does incorporate changes made by programs, but not the parts of the program which do *not* change. In [16], this has been pointed out. This work introduces explicit frames to keep track of all program variables which occur within a program.

Definition 16 (Program frame). *For a program π let $\boldsymbol{\xi}_\pi$ denote the finite vector of program variables (local or global) syntactically appearing in π, called the* frame *of π.*

The finiteness of $\boldsymbol{\xi}_\pi$ raises from the fact that variables must occur syntactically. Where the program π is clear, we omit it.

In [16], a characteristic function for a program π w.r.t. two frames $\boldsymbol{\xi}$ and $\boldsymbol{\xi}'$ (initial and final) has been introduced. The elements of the frames are free variables which occur in the formula representing pre-execution and post-execution values. The vector $\boldsymbol{\xi}'$ contains fresh variables which correspond to those in $\boldsymbol{\xi}$ with the difference of being primed. We lift this two-state technique to a setting were the formula contains a frame for each state on the trace of π. Since there might be an infinite number of them, we introduce binary function symbol *val* over natural numbers with the intuitive interpretation that $val(m, n)$ denotes the value of the m-th variable (w.r.t. the ordering in $\boldsymbol{\xi}$) in the n-th intermediate state. Let i_v denote the index of a variable v in a given variable vector.

Not every intermediate state is in the trace of π, however. Therefore we introduce a unary predicate *step* where $step(n)$ indicates that state n is on the trace. Both *val* and *step* are definable in DTL.

As we have defined them, traces do not contain intermediate steps. In order to axiomatize *val* and *step* w.r.t. a given program π, we have to find a program π' whose trace is a refinement of the one of π which includes every intermediate state. It is easy to see that this can achieved by adding global nop assignments: For a given program π, let the program π' coincide with π except that after each statement which is not a global assignment the assignment `Z = Z;` occurs (where `Z` is a global program variable not occurring in π). Then, *val* can be

axiomatized through the following formula (where x is an expression without program variables):

$$val(i_v, n) \doteq x \leftrightarrow [\![\pi']\!] \underbrace{\circ \cdots \circ}_{n} v \doteq x$$

For an index n in the trace of π', let n^* denote the index in the trace of the original program π such that $trc(\pi)[n^*]$ is the next state which occurs in both traces. Then, we can axiomatize *step* through the following formula:

$$step(n) \leftrightarrow ([\![\pi']\!] \underbrace{\circ \cdots \circ}_{n} \phi \leftrightarrow [\![\pi]\!] \underbrace{\circ \cdots \circ}_{n^*} \phi)$$

We now define the characteristic formula $\mathcal{T}(\pi, n, n')$ with the understanding that when program π is executed in the state encoded by n, n' encodes a terminal state. The base case is the empty program ϵ, which states that the current state is terminal.

$$\mathcal{T}(\epsilon, n, n') := n \doteq n'$$

$$\mathcal{T}(\texttt{v:= x;}\, \omega, n, n') := val(i_v, n+1) \doteq x^* \wedge \textstyle\bigwedge_{j \neq i_v} val(j, n+1) \doteq val(j, n) \\ \wedge n' \doteq n+1 \wedge \mathcal{T}(\omega, n+1)$$

The expression x^* is produced from x by replacing every program variable p by $val(i_p, n)$. For global variables, $\mathcal{T}$ looks similar, with the exception that $step(n)$ is included.

$$\mathcal{T}(\texttt{G:= x;}\, \omega, n, n') := val(i_G, n+1) \doteq x^* \wedge \textstyle\bigwedge_{j \neq i_G} val(j, n+1) \doteq val(j, n) \\ \wedge step(n) \wedge n' \doteq n+1 \wedge \mathcal{T}(\omega, n+1)$$

In the case of a conditional, $\mathcal{T}$ consists of a case distinction on the condition b. We also have to distinguish the cases whether there is a final state n'' of π_j (meaning that π_j terminates) or not, in which case the remaining program ω has no effect on the resulting formula.

$$\mathcal{T}(\texttt{if (b) \{}\pi_1\texttt{\} else \{}\pi_2\texttt{\}}\ \omega, n, n') := \\ (b^* \rightarrow (\mathcal{T}(\pi_1, n, n') \vee \exists n''.(\mathcal{T}(\pi_1, n, n'') \wedge \mathcal{T}(\omega, n'', n')))) \\ \wedge(\neg b^* \rightarrow (\mathcal{T}(\pi_2, n, n') \vee \exists n''.(\mathcal{T}(\pi_2, n, n'') \wedge \mathcal{T}(\omega, n'', n'))))$$

The formula $\mathcal{T}$ for the case of a loop is not displayed here for its immense complexity. It introduces new function symbols to encode the number of iterations and a formula which is defined through repeated axiomatizations, which uses a Gödelization of the loop. The nevertheless interested reader may refer to [16, Sect. 4.5.2], where a simpler version is used to show completeness of the two-state-based version of $\mathcal{T}$. The basic ideas behind this construction apply to our setting, too, since $\mathcal{T}$ only encodes program semantics and is independent of other logic operators. As an important feature of this construction, it is well-founded since the recursive definition of $\mathcal{T}$ ranges over the statement length of the program.

B.2 Embedding of DTL Into First-order Logic

In the following, we define a first-order formula $\mathcal{F}(\phi)$ for every DTL formula ϕ which does not contain updates or program modalities. It contains two free variables n and n', which, respectively, encode the current state in the trace and the final state. In the case that the program does not terminate, the formula $\mathcal{T}$ as constructed above does not restrict n' in any way, i.e., it is implicitly universally quantified. A formula $\forall n''.n'' < n'$ then essentially amounts to "forall natural numbers".

In a boolean expression, the occurring variables are replaced by *var* terms; for propositional connectives or quantifiers, the transformation denoted by $\mathcal{F}$ is just propagated. In the case of "throughout", $\mathcal{F}(\Box\phi)$ states that in every following stepping state $\mathcal{F}$ holds. Similarly, $\mathcal{F}(\phi \mathbf{U} \psi)$ states that there is a stepping state such that both $\mathcal{F}(\psi)$ and until that $\mathcal{F}(\Box\phi)$ holds. For "weak next" ($\bullet$), the formula states that the trace ends in n or in the next stepping state n^+, $\mathcal{F}(\phi)$ holds.

Definition 17. *Let e be an expression and let ϕ, ϕ_1, and ϕ_2 be formula. The formula $\mathcal{F}$ (with two free variables n and n') is constructed as follows:*

$$
\begin{array}{ll}
\mathcal{F}(e, n, n') & := e^* \\
\mathcal{F}(\neg\phi, n, n') & := \neg\mathcal{F}(\phi, n, n') \\
\mathcal{F}(\phi_1 \wedge \phi_2, n, n') & := \mathcal{F}(\phi_1, n, n') \wedge \mathcal{F}(\phi_2, n, n') \\
\mathcal{F}(\forall x.\phi, n, n') & := \forall x.\mathcal{F}(\phi, n, n') \\
\mathcal{F}(\Box\phi, n, n') & := \forall n''.(n \leq n'' < n' \wedge \mathit{step}(n'') \rightarrow \mathcal{F}(\phi, n'', n')) \\
\mathcal{F}(\phi \mathbf{U} \psi, n, n') & := \exists n''.(n \leq n'' < n' \wedge \mathit{step}(n'') \wedge \mathcal{F}(\psi, n'', n') \wedge \mathcal{F}(\Box\phi, n, n'')) \\
\mathcal{F}(\bullet\phi, n, n') & := n \doteq n' \vee \forall n^+.(n^+ > n \wedge \mathit{step}(n^+) \\
 & \qquad \wedge \forall n''.(n \leq n'' < n^+ \rightarrow \neg \mathit{step}(n'')) \rightarrow \mathcal{F}(\phi, n^+, n'))
\end{array}
$$

In the case of a boolean expression e, we obtain e^* through replacing every program variable p with the term $\mathit{val}(i_p, n)$.

For formulae containing updates we have to take good care of the cases where other updates might occur deeper in the formula. For instance, it is unsound to rewrite a formula $\{v := x\}\phi$ to $\phi[v/x]$; take $\phi = \{v := x + 1\}(v \doteq x)$ for an obvious conflict. We therefore adopt the notion of *parallel updates* [18]. As the name suggests, parallel updates contain assignments to several variables, which are independent of each other. I.e., variables may only occur once on the left-hand side. A parallel update $\{v_1 := x_1 \| \ldots \| v_k := x_k\}$ thus has the same semantics as $\{v_1 := x_1\} \cdots \{v_k := x_k\}$, but the other direction does not hold. Due to those properties, parallel updates can be used as a normal form for updates.

Ordinary updates are parallel by definition. For sequences of updates, we use the following procedure to parallelize them: Let $\mathcal{U} = \{v_1 := x_1 \| \ldots \| v_k := x_k\}$ be a parallel update. The parallelized counterpart of $\mathcal{U}\{v := x\}$ then is either

- $\{v_1 := x_1 \| \ldots \| v_k := x_k \| v := x[v_1/x_1, \ldots, v_k/x_k]\}$ in case that v does not occur on the left-hand side in $\mathcal{U}$, or

– $\{v_1 := x_1 \| \ldots \| v_{l-1} := x_{l-1} \| v_l := x[v_1/x_1, \ldots, v_l/x_l, \ldots, v_k/x_k] \| \ldots \| v_k := x_k\}$ in case that $v = v_l$.

Note that v may occur on the right-hand side of $\mathcal{U}$; this is not a conflicting case. We are now able to replace any sequence of updates $\mathcal{U}$ by a parallelized version $\mathcal{U}^{\|}$.

Let ϕ be formula which is not prefixed by updates or a program modality and let $\mathcal{U}$ be sequence of updates. We define

$$\mathcal{F}(\mathcal{U}\phi, n, n') := \mathcal{F}(\phi[v_1 \sharp x_1^*, \ldots, v_k \sharp x_k^*], n, n')$$

where $\{v_1 := x_1 \| \ldots \| v_k := x_k\} = \mathcal{U}^{\|}$. The use of weak substitutions means that deeper inside the formula ϕ new sequences of ordinary updates may appear. Since updates are either eliminated (as substitutions may be applied) or further pushed in, this recursive definition of $\mathcal{F}$ is well-founded. However, we still need to define $\mathcal{F}$ for the case in which ϕ has the shape $[\![\pi]\!]\phi'$.

In order to define $\mathcal{F}$ for a formula with a program, we need the characteristic formula $\mathcal{T}$ and a first-order formula describing the trace property. In the special case without updates, it would like the following:

$$\mathcal{F}([\![\pi]\!]\phi, n, n') := \mathcal{T}(\pi, n'', n''') \wedge \mathcal{F}(\phi, n'', n''')$$

The program π gives rise to a completely new trace; in order to avoid any naming conflicts, this trace starts in a state n'' and possibly terminates in state n''' where n'' and n''' are fresh variables. Note that we do not require something like $n'' > n'$ since n'' is implicitly universally quantified. Remember that we have defined program variables to not have an initial value. Instead, initial values are imposed using updates:

$$\mathcal{F}(\mathcal{U}[\![\pi]\!]\phi, n, n') := \mathcal{T}(\pi, n'', n''') \wedge \mathcal{F}(\phi, n'', n''') \wedge \bigwedge_{1 \leq j \leq k} val(i_{v_j}, n'') \doteq x_j^*$$

where $\mathcal{U}^{\|} = \{v_1 := x_1 \| \ldots \| v_k := x_k\}$, i_{v_j} is the index of v_j in $\boldsymbol{\xi}_\pi$, and x_j^* is produced from x_j by replacing every variable v by $val(i_v, n)$ as introduced above. This covers the final case in the definition of $\mathcal{F}$. We are now able to state the following fundamental lemmas, from which it follows Lemma 1.

Lemma 20. *The formulae $\mathcal{F}(\cdot,\cdot,\cdot)$ and $\mathcal{T}(\cdot,\cdot,\cdot)$ are well-defined formulae of first-order logic with arithmetic.*

Following the above definitions of $\mathcal{F}$ and $\mathcal{T}$, we can show this lemma by induction over the length of a program or a formula (i.e., the number of logical connectives), respectively.

Lemma 21 (Expressibility of trace formulae). *Let s be state, π be a program, ψ be a DTL formula, and $n \in \mathbb{N}$. Then*

$$trc(s, \pi)[n, \infty) \vDash \psi \leftrightarrow (\mathcal{T}(\pi, 0, n') \wedge \mathcal{F}(\psi, n^*, n'))$$

where n^* encodes the n-th stepping state in $trc(s, \pi)$.

Intuitively, this lemma states that the program trace of π is represented by the formula $\mathcal{T}(\pi, 0, n')$ where the inital state s is represented by the number 0 and on the subtrace beginning in state n^* (i.e., the state reached after n temporal steps), the FOL representation $\mathcal{F}(\psi)$ of ψ holds. Therefore, the formula $\phi_{\psi,s,\pi,n}$ which we were looking for in Lemma 1 is exactly $\mathcal{T}(\pi, 0, n') \wedge \mathcal{F}(\psi, n^*, n')$. The above definitions of $\mathcal{T}$ and $\mathcal{F}$ can be understood as a constructive proof to this lemma.

Since the above two lemmas state that the semantics of programs can be given in first-order arithmetic precisely, as a corollary, we are able to assert the existence of invariants and variants—in particular the *strongest* invariant or variant. We will use this property in the following section to show that every valid formula of the form $[\![\texttt{while (b) \{}\pi\texttt{\}}\ \omega]\!]\phi$ is derivable in $\mathcal{C}_{\mathrm{DTL}}$.

B.3 Derivability

In this section, we finally show that any valid DTL formula is derivable in $\mathcal{C}_{\mathrm{DTL}}$ (within a finite number of steps). Although in Sect. B.2, we have shown that to every DTL formula there is a logically equivalent FOL formula, we still need to prove that our calculus can handle them. The proof essentially amounts to showing that for every valid formula there exists a possible rule application which "brings the proof forward".

At first, we need to formalize this "bringing forward". Clearly, the proof is brought forward if the formulae in the premisses are less complex than in the conclusion. Still, this complexity has many facets because of the interplay of different logical operators, i.e., propositional, quantifiers, updates, program modalities, and temporal operators.

Definition 18 (Complexity measure). *For a formula ϕ, we define the following measures $\chi_\cdot(\phi) \in \mathbb{N}$:*

- Temporal progressiveness $\chi_T(\phi)$ *is the sum of (i) the number of subformulae of the shape $[\![\texttt{G = a;}\ \omega]\!]\psi$ where the top-level operator of ψ is not a 'next' operator and (ii) the sum ranging over all subformulae of the shape $[\![\pi]\!] \circledcirc \psi$ (where $\circledcirc$ stands for either weak or strong next) of the maximum number of statements preceding the first global assignment on all branches (w.r.t. splits induced by* `while` *and* `if`*) in π (or the total statement count if there is no assignment in π).*
- Program length $\chi_P(\phi)$ *is the total number of statements in all programs occurring in ϕ; the statement length of a single program π will be denoted by $|\pi|$.*
- Negation height $\chi_N(\phi)$ *is the number of logical operators within the scope of a negation.*
- Update height $\chi_U(\phi)$*: is the number of logical operators within the scope of an update.*
- Formula complexity $\chi_F(\phi)$ *is the total number of logical operators.*

The total complexity *measure χ is the vector $(\chi_T, \chi_P, \chi_N, \chi_U, \chi_F) \in \mathbb{N}^5$. The ordering $<$ is to be understood lexicographically and the we define addition on total measures component-wise.*

The definition of χ_T may seem a bit arbitrary at first sight, but we need to define a measure under which rules R20 ("unwind box") and R27 ("loop unwind") constitute a progress in the proof. Although they lead to more complex formulae in the sense of the other measures, actual progress relies on their interplay with rule R25 ("global assignment"), which is only applicable whenever the top-level operator on the trace is weak or strong next.

As an example for the complexity measures, take the following formulae ϕ and ϕ':

$$\begin{aligned}\phi &= \neg(\mathcal{U} b \wedge [\![\texttt{while (b) \{X = X+1;\}}]\!]\bullet\Diamond\neg b)\\ \phi' &= \neg\mathcal{U} b \vee \neg[\![\texttt{while (b) \{X = X+1;\}}]\!]\bullet\Diamond\neg b\end{aligned}$$

The measures of ϕ are $\chi_T(\phi) = 1$, $\chi_P(\phi) = 2$, $\chi_N(\phi) = 6$, $\chi_U(\phi) = 0$, and $\chi_F(\phi) = 7$. Obviously, the logically equivalent formula ϕ' has more operators in total—it is $\chi_F(\phi') = 8$, but the negation is delegated which leads to $\chi_N(\phi') = 4$ and by the lexicographical ordering, $\chi(\phi') < \chi(\phi)$.

Definition 19 (Progress). *Let ϕ be a valid formula. A rule*

$$\frac{\Gamma_1 \vdash \Delta_1 \quad \cdots \quad \Gamma_n \vdash \Delta_n}{\vdash \phi}$$

is progressing *if for every branch (over the index $i \in [1, n]$) there is a formula $\phi' \in \Gamma_i \cup \Delta_i$ with $\chi(\phi') < \chi(\phi)$.*

Lemma 22. *For any valid formula there is a progressing rule in $\mathcal{C}_{\mathrm{DTL}}$.*

The few rules in the calculus which are not progressing are not essential to the completeness proof. Remember that we only need to show the derivability of a sequent $\vdash \phi$ where ϕ is a valid formula. Rules R23 and R32 are additional in the sense that their removal from the calculus would not threaten completeness.

Proof. Since we have constructed the definition of χ with the intention of showing Lemma 22, we will not give a proof for every rule here, but only for the more interesting cases, i.e., rules R20 ("unwind box"), R27 ("unwind loop") and R28 ("loop invariant box").

R20: For the formula $\phi_0 = \mathcal{U}[\![\pi]\!]\Box\phi$ in the conclusion, assume that the first statement in π is a global assignment (otherwise either rule R24, R26, or R28 would be applicable). It is $\chi(\phi_0) = (1, |\pi|, 0, 2, 3) + \chi(\phi)$. For the formulae $\phi_1 = \mathcal{U}[\![\pi]\!]\bullet\Box\phi$ and $\phi_2 = \mathcal{U}[\![\pi]\!]\phi$ in the premisses, it is $\chi(\phi_1) = (0, |\pi|, 0, 2, 4) + \chi(\phi)$ and $\chi(\phi_2) = (1, |\pi|, 0, 2, 2) + \chi(\phi)$, thus $\chi(\phi_1) < \chi(\phi_0)$ and $\chi(\phi_2) < \chi(\phi_0)$.

R27: Assume the formula $\phi_0 = \mathcal{U}[\![\lambda\,\omega]\!]\circledcirc\phi$ in the conclusion with $\lambda = \texttt{while (b) \{}\pi\texttt{\}}$. (If there were another operator instead of a 'next' as the top-level operator of the trace formula, other rules could apply). It is $\chi(\phi_0) = (d, |\lambda| + |\omega|, 0, 2, 3) +$

$\chi(\phi)$, where $d \in \mathbb{N}$ is the maximum length of statements before the first occurrence of global assignment statement. Take the formula $\phi_1 = \mathcal{U}[\![\pi\ \lambda\ \omega]\!] \circledcirc \phi$ in the first premiss. By Def. 2, π contains at least one global assignment. Therefore, it is $\chi(\phi_1) = (d-1, |\pi|+|\lambda|+|\omega|, 0, 2, 3)+\chi(\phi)$. For $\phi_2 = \mathcal{U}[\![\omega]\!] \circledcirc \phi$ in the second premiss, it is $\chi(\phi_2) = (d, |\omega|, 0, 2, 3) + \chi(\phi)$.

R28: In this case, the formulae in the premisses were obviously less complex if it were not for the invariant *Inv*. Since, however, through Lemma 1, we have shown that the strongest invariant can be given as a non-temporal formula, the formulae in the premisses of rule R28 have the same χ_T as the conclusion and strictly less χ_P. △

Finally, we are able to show that any valid formula ϕ is derivable within $\mathcal{C}_{\mathrm{DTL}}$. From Lemma 22, we draw that there exists an exhaustive proof strategy, i.e., a sequence of rule applications such that there is a proof tree rooted in $\vdash \phi$ such that there are no more rules applicable (or at least some kind of fix-point is reached where every applicable rule has been applied before on the same formula[4]). What is still left to show is that the leaves of an exhausted proof tree are the empty sequent, i.e., validity of ϕ is justified by axioms.

Definition 20 (Local completeness). *A rule is (locally)* complete *if the following holds: If the conclusion is valid, then all premisses are valid.*

Local completeness of a rule is the dual property to soundness of a rule. There may be rules which are both sound and progressing, but not complete. A typical example is a 'hiding' rule like $\dfrac{\vdash \phi}{\vdash \phi \vee \psi}$. If ψ is valid, so is $\phi \vee \psi$, but this is not derivable.

Lemma 23. *For any valid formula ϕ, there is a complete and progressing rule in $\mathcal{C}_{\mathrm{DTL}}$.*

$\downarrow \pi \quad \phi \rightarrow$	e	$\phi_1 \wedge \phi_2$	$\phi_1 \vee \phi_2$	$\forall x.\phi'$	$\Box\phi'$	$\Diamond\phi'$	$\phi_1 \mathbf{U} \phi_2$	$\bullet\phi'$	$\circ\phi'$
ϵ	R16	R10	R9	R17	R20	R21	R19	R22	R23
`G = a;`	R16	R10	R9	R17	R20	R21	R19	R25	R25
`l = a;`	R16	—	—	—	R24	—	—	—	—
`if...`	R16	—	—	—	R26	—	—	—	—
`while...`	R16	R10	R9	R17	R28	R29	R30	R27	R27

$\phi \rightarrow$	$\neg e$	$\neg(\phi_1 \wedge \phi_2)$	$\neg(\phi_1 \vee \phi_2)$	$\neg(\forall x.\phi')$	$\neg\Box\phi'$	$\neg\Diamond\phi'$	$\neg(\phi_1 \mathbf{U} \phi_2)$	$\neg\bullet\phi'$	$\neg\circ\phi'$
π	R16	n/a	R14	R18	n/a	R14	R12	n/a	R14

Table 8. Rules for formulae of the shape $\mathcal{U}[\![\pi\ \omega]\!]\phi$. Some of the negation rules do not exist since the formula is replaced by its dual.

[4] This typically happen through the use of γ rules, which instantiate a quantifier but keep the original quantified formula.

Proof. We sketch the proof here. For rules related to propositional operators and quantifiers, complete such proofs can be found in standard textbooks. Table 8 shows the corresponding rules for formulae of the shape $\mathcal{U}[\![\pi\ \omega]\!]\phi$. The soundness proofs in Sect. A for each of them can actually be read in the opposite direction and therefore are equivalence proofs. Most of those are quite obvious. In the cases of invariant rules, especially in the respective final premiss, the use case, this is not so obvious. Essential to proving this case is Lemma 1, from which it follows that the *strongest* invariant exists, while a weaker one, for instance *true*, would not suffice.

Theorem 2 is a direct corollary from Lemma 23, since we have shown that for every valid formula there is a finite sequence of rule applications which result in a closed proof.

References

1. Martín Abadi and Zohar Manna. Nonclausal temporal deduction. In Rohit Parikh, editor, *Logic of Programs*, volume 193 of *LNCS*, pages 1–15, Brooklyn, NY, June 1985. Springer.
2. Martín Abadi and Zohar Manna. Nonclausal deduction in first-order temporal logic. *Journal of the ACM*, 37(2):279–317, April 1990.
3. Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing LTL semantics for runtime verification. *J. Log. Comput*, 20(3):651–674, 2010.
4. Bernhard Beckert. *Tableau-based Theorem Proving: A Univied View. Integrating and Unifying Methods of Tableau-based Theorem Proving.* PhD thesis, Universität Karlsruhe. Department of Computer Science, 1998.
5. Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2007.
6. Bernhard Beckert and André Platzer. Dynamic logic with non-rigid functions: A basis for object-oriented program verification. In Ulrich Furbach and Natarajan Shankar, editors, *Proceedings, International Joint Conference on Automated Reasoning, Seattle, USA*, LNCS 4130, pages 266–280. Springer, 2006.
7. Bernhard Beckert and Steffen Schlager. A sequent calculus for first-order dynamic logic with trace modalities. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Proceedings, International Joint Conference on Automated Reasoning, Siena, Italy*, LNCS 2083, pages 626–641. Springer, 2001.
8. Marco Benedetti and Alessandro Cimatti. Bounded model checking for past LTL. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2619 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2003.
9. Rayna Dimitrova, Bernd Finkbeiner, Máté Kovács, Markus N. Rabe, and Helmut Seidl. Model checking information flow in reactive systems. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, January 2012.
10. Martin Giese. A calculus for type predicates and type coercion. In Bernhard Beckert, editor, *TABLEAUX*, volume 3702 of *Lecture Notes in Computer Science*, pages 123–137. Springer, 2005.
11. Rajeev Goré. Tableau methods for modal and temporal logics. In Marcello D'Agostino, Dov Gabbay, Reiner Hähnle, and Joachim Posegga, editors, *Handbook of Tableau Methods*, pages 297–396. Kluwer Academic Publishers, Dordrecht, 1999.
12. David Harel. *First-order dynamic logic*, volume 68 of *Lecture notes in computer science*. Springer-Verlag, New York, 1979.
13. David Harel. Dynamic logic. In Dov Gabbay and Franz Guenther, editors, *Handbook of Philosophical Logic, Volume II: Extensions of Classical Logic*, volume 2, pages 497–604. D. Reidel Publishing Co., Dordrecht, 1984.
14. Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.
15. Ben Moszkowski. A temporal logic for multilevel reasoning about hardware. *IEEE Computer*, 18(2), February 1985.
16. André Platzer. An object-oriented dynamic logic with updates. Master's thesis, Universität Karlsruhe, 2004.

17. Mark Reynolds and Clare Dixon. Theorem-proving for discrete temporal logic. In *Handbook of temporal reasoning in artificial intelligence*. Elsevier Science, 2005.
18. Philipp Rümmer. Sequential, parallel, and quantified updates of first-order structures. In *Logic for Programming, Artificial Intelligence and Reasoning*, volume 4246 of *Lecture Notes in Computer Science*, pages 422–436. Springer, 2006.
19. Gerhard Schellhorn, Bogdan Tofan, Gidon Ernst, and Wolfgang Reif. Interleaved programs and rely-guarantee reasoning with ITL. In Carlo Combi, Martin Leucker, and Frank Wolter, editors, *TIME*, pages 99–106. IEEE, 2011.
20. Andreas Thums, Gerhard Schellhorn, Frank Ortmeier, and Wolfgang Reif. Interactive verification of statecharts. In Hartmut Ehrig, Werner Damm, Jörg Desel, Martin Große-Rhode, Wolfgang Reif, Eckehard Schnieder, and Engelbert Westkämper, editors, *Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *Lecture Notes in Computer Science*, pages 355–373. Springer, 2004.
21. Pierre Wolper. The tableau method for temporal logic: An overview. *Logique et Analyse*, 28(110–111):119–136, June–September 1985.