A Sequent Calculus for First-order Dynamic Logic with Trace Modalities

Bernhard Beckert and Steffen Schlager

University of Karlsruhe
Institute for Logic, Complexity and Deduction Systems
D-76128 Karlsruhe, Germany
beckert@ira.uka.de, schlager@ira.uka.de

Abstract. The modalities of Dynamic Logic refer to the final state of a program execution and allow to specify programs with pre- and post-conditions. In this paper, we extend Dynamic Logic with additional trace modalities "throughout" and "at least once", which refer to *all* the states a program reaches. They allow to specify and verify invariants and safety constraints that have to be valid throughout the execution of a program. We give a sound and (relatively) complete sequent calculus for this extended Dynamic Logic.

1 Introduction

We present a sequent calculus for an extended version of Dynamic Logic (DL) that has additional modalities "throughout" and "at least once" referring to the intermediate states of program execution.

Dynamic Logic [9, 5, 8] can be seen as an extension of Hoare logic [2]. It is a first-order modal logic with modalities $[\alpha]$ and $\langle\alpha\rangle$ for every program α. These modalities refer to the worlds (called states in the DL framework) in which the program α terminates when started in the current world. The formula $[\alpha]\phi$ expresses that ϕ holds in *all* final states of α, and $\langle\alpha\rangle\phi$ expresses that ϕ holds in *some* final state of α. In versions of DL with a non-deterministic programming language there can be several such final states (worlds). Here we consider a Deterministic Dynamic Logic (DDL) with a deterministic *while* programming language [4, 6]. For deterministic programs there is exactly one final world (if α terminates) or there is no final world (if α does not terminate). The formula $\phi \to \langle\alpha\rangle\psi$ is valid if, for every state s satisfying pre-condition ϕ, a run of the program α starting in s terminates, and in the terminating state the post-condition ψ holds. The formula $\phi \to [\alpha]\psi$ expresses the same, except that termination of α is not required, i.e., ψ must only holf *if* α terminates.

Thus, $\phi \to [\alpha]\psi$ is similar to the Hoare triple $\{\phi\}\alpha\{\psi\}$. But in contrast to Hoare logic, the set of formulas of DL is closed under the usual logical operators. In Hoare logic, the formulas ϕ and ψ are pure first-order formulas, whereas in DL they can contain programs. That is, DL allows to involve programs in the formalisation of pre- and post-conditions. The advantage of using programs is that one can easily specify, for example, that some data structure is not cyclic, which is impossible in pure first-order logic.

In some regard, however, standard DL (and DDL) is still lacking expressivity: The semantics of a program is a relation between states; and formulas can only be used to describe the input/output behaviour of programs. Standard DL cannot be used to reason about program behaviour not manifested in the input/output relation. It is inadequate for reasoning about non-terminating programs, and it cannot be used to verify invariants or safety constraints that have to be valid throughout program execution.

We overcome this deficiency and increase the expressivity of DDL by adding two new modalities $[\![\alpha]\!]$ ("throughout") and $\langle\!\langle\alpha\rangle\!\rangle$ ("at least once"). In the extended logic, which we call (Deterministic) Dynamic Logic with Trace Modalities (DLT), the semantics of a program is the sequence of all states its execution passes through when started in the current state (its *trace*). It is possible in DLT to specify properties of the intermediate states of terminating and non-terminating programs. And such properties, which are typically safety constrains, can be verified using the calculus presented in Section 4. This is of great importance as safety constraints occur in many application domains of program verification (the simplest type of such constraints is that the value of a variable must never get out of certain bounds).

Previous work in this area includes Pratt's Process Logic [9, 10], which is an extension of *propositional* DL with trace modalities (DLT can be seen as a first-order Process Logic). Also, Temporal Logics have modalities that allow to talk about intermediate states. In Temporal Logics, however, the program is fixed and considered to be part of the structure over which the formulas are interpreted. Temporal Logics, therefore, do not have the compositionality of Dynamics Logics.

The calculus for DDL described in [6] (which is based on the one given in [4]) has been implemented in the software verification systems KIV [11] and VSE [7]. It has successfully been used in practice to verify software systems of considerable size.

The work reported here has been carried out as part of the KeY-Projekt [1].[1] The goal of KeY is to enhance a commercial CASE tool with functionality for formal specification and deductive verification and, thus, to integrate formal methods into real-world software development processes. In the KeY project, a version of DL for the Java Card programming language [3] is used for verification. Deduction in DL (and DLT) is based on symbolic program execution and simple program transformations and is, thus, close to a programmer's understanding of a program's semantics. Our motivation for considering trace modalities was that in typical real-world specifications as they are done with the help of CASE tools, there are often program parts for which invariants and safety constraints are given, but for which the user did not bother to give a full specification with pre- and post-conditions.

The structure of this paper is as follows: The syntax of DLT is defined in Section 2 and its semantics in Section 3. In Section 4, we describe our sequent calculus for DLT. Theorems stating soundness and (relative) completeness are presented in Section 5 (due to space restrictions, the proofs are only sketched, they can be found in [12]). In Section 6, we give an example for verifying that a non-terminating program preserves a certain invariant. Finally, in Section 7, we discuss future work.

2 Syntax of DL with Trace Modalities

In first-order DL, states are not abstract points (as in propositional DL) but valuations of variables. Atomic programs are assignments of the form x := t. Executing x := t changes the program state by assigning the value of the term t to the variable x. The value of a term t depends on the current state s (namely the value that s gives to the variables occurring in t). The function symbols are interpreted using a fixed first-order structure. This *domain of computation*, over which quantification is allowed, can be considered to define the data structures used in the programs. The logic DLT as well as the calculus presented in Section 4 are basically independent of the domain actually used. The only restriction is

[1] More information on KeY can be found at `i12www.ira.uka.de/~key`.

that the domain must be sufficiently expressive. In the following, for the sake of simplicity, we use arithmetic as the single domain. In practice, there will be additional function and predicate symbols and different types of variables ranging over different sorts of a many-sorted domain (different data structures). Equality $\doteq$ must be defined on each type.

The arithmetic *signature* $\Sigma_{\mathbb{N}}$ contains

- the constant 0 (zero) and the unary function symbol s (successor) as constructors (in the following we abbreviate terms of the form $s(\cdots s(0)\cdots)$ with their decimal representation, e.g. "2" abbreviates "$s(s(0))$"),
- the binary function symbols $+$ (addition) and $*$ (multiplication), and
- the binary predicate symbols $\leq$ (less or equal than) and $\doteq$ (equality).

In addition, there is an infinite set *Var* of *object variables*, which are also used as program variables. The set $\mathit{Term}_{\mathbb{N}}$ of *terms* over $\Sigma_{\mathbb{N}}$ is built as usual in first-order predicate logic (FOL) from the variables in *Var* and the function symbols in $\Sigma_{\mathbb{N}}$.

The syntax of DLT-formulas is defined in three steps. First, we define—in the usual way—the set FOL-formulas, i.e., formulas of first-order predicate logic without modal operators (Def. 1). Then we define what the programs of the deterministic programming language of DDL and DLT are (Def. 2). They contain FOL-formulas as conditions in if-then-else and loop statements. The last step is to define the formulas of full DLT (Def. 3).

We use the classical connectives $\wedge$ (conjunction), $\vee$ (disjunction), $\rightarrow$ (implication), and $\neg$ (negation), and the quantifier symbols $\forall$ and $\exists$.

Definition 1. *The set of FOL-formulas is recursively defined by:*

- *true and false are FOL-formulas.*
- *If $t_1, t_2 \in \mathit{Term}_{\mathbb{N}}$, then $t_1 \leq t_2$ and $t_1 \doteq t_2$ are (atomic) FOL-formulas.*
- *If ϕ, ψ are FOL-formulas, then so are $\neg\phi$, $\phi \vee \psi$, $\phi \wedge \psi$, and $\phi \rightarrow \psi$.*
- *If ϕ is an FOL-formula and $x \in \mathit{Var}$, then $\exists x\, \phi$ and $\forall x\, \phi$ are FOL-formulas.*

The programming constructs for forming the complex programs of DDL and DLT from the atomic assignments are the concatenation of programs, if-then-else conditionals, and while loops.

Definition 2. *The set of programs of DLT is recursively defined by:*

- *If $x \in \mathit{Var}$ and $t \in \mathit{Term}_{\mathbb{N}}$, then x `:=` t is a program (assignment).*
- *If α and β are programs, then α`;`β is a program (concatenation).*
- *If α and β are programs and ϵ is a quantifier-free FOL-formula (Def. 1), then* `if` ϵ `then` α `else` β *is a program (conditional).*
- *If α is a program and ϵ is a quantifier-free FOL-formula (Def. 1), then* `while` ϵ `do` α *is a program (loop).*

The programs of DLT form a computationally complete programming language. For every partial recursive function $f : \mathbb{N} \rightarrow \mathbb{N}$ there is a program $\alpha_f(x)$ that computes f, i.e., if $\alpha_f(x)$ is started in an arbitrary state in which the value of x is some $n \in \mathbb{N}$, then it terminates in a state in which the value of x is $f(n)$.

We now proceed to define the formulas of DLT. Note, that the first four conditions in Definition 3 are identical to those in the definition of FOL-formulas (Def. 1). Only the last condition is new, which adds the modalities (and programs) to the formulas.

Definition 3. *The set of DLT-formulas is recursively defined by:*

- *true and false are DLT-formulas.*
- *If $t_1, t_2 \in \mathit{Term}_{\mathbb{N}}$, then $t_1 \leq t_2$ and $t_1 \doteq t_2$ are (atomic) DLT-formulas.*

– *If ϕ, ψ are DLT-formulas, then so are $\neg\phi$, $\phi \lor \psi$, $\phi \land \psi$, and $\phi \to \psi$.*
– *If ϕ is a DLT-formula and $x \in Var$, then $\exists x\, \phi$, $\forall x\, \phi$ are DLT-formulas.*
– *If ϕ is a DLT-formula and α is a program (Def. 2), then $[\alpha]\phi$, $\langle\alpha\rangle\phi$, $[\![\alpha]\!]\phi$, and $\langle\!\langle\alpha\rangle\!\rangle\phi$ are DLT-formulas.*

Definition 4. *A* sequent *is of the form $\phi_1, \dots, \phi_m \vdash \psi_1, \dots, \psi_n$ $(m, n \geq 0)$, where the ϕ_i and ψ_j are DLT-formulas. The order of the ϕ_i resp. the ψ_j is irrelevant, i.e., $\phi_1, \dots, \phi_m$ and $\psi_1, \dots, \psi_n$ are treated as multi-sets.*

In first-order DL, not only quantifiers but also modalities can bind variables.

Definition 5. *A variable $x \in Var$ is* bound *in a DLT-formula ϕ if it occurs inside the scope of (i) a quantification $\forall x$ resp. $\exists x$, or (ii) a modality $[\alpha]$, $\langle\alpha\rangle$, $[\![\alpha]\!]$, or $\langle\!\langle\alpha\rangle\!\rangle$ containing an assignment $x := t$. The variable x is* free *in ϕ if there is an occurrence of x in ϕ that is neither bound by a quantifier nor a modality.*

Definition 6. *A* substitution *assigns to each object variable in Var a term in $Term_{\mathbb{N}}$. A substitution σ is applied to a DLT-formula ϕ by replacing all free occurrences of variables x in ϕ by $\sigma(x)$.*

If a substitution $\{x/t\}$ instantiates only a single variable x, its application to a formula ϕ or a formula sequence Γ is denoted by ϕ^t_x resp. Γ^t_x.

A substitution σ is admissible *w.r.t. a DLT-formula ϕ if there are no variables x and y such that x is free in ϕ, y occurs in $\sigma(x)$, and, after replacing $\sigma(x)$ for some free occurrence of x in ϕ, the occurrence of y in $\sigma(x)$ is bound in $\phi\sigma$.*

3 Semantics of DL with Trace Modalities

Since we use arithmetic as the only domain of computation, the semantics of DLT is defined using a single fixed model, namely $\langle \mathbb{N}, I_{\mathbb{N}} \rangle$. It consists of the universe $\mathbb{N}$ of natural numbers and the canonical interpretation function $I_{\mathbb{N}}$ assigning the function and predicate symbols of $\Sigma_{\mathbb{N}}$ their natural meaning in arithmetic.

The states (worlds) of the model (only) differ in the value assigned to the object variables. Therefore, the states can be defined to *be* variable assignments.

Definition 7. *A* state *s assigns to each variable $x \in Var$ a number $s(x) \in \mathbb{N}$.*

Let $x \in Var$ and $n \in \mathbb{N}$; then $s' = s\{x \leftarrow n\}$ is the state that is identical to s except that x is assigned n, i.e., $s'(x) = n$ and $s'(y) = s(y)$ for all $x \neq y$.

To define the semantics of DLT-formulas (Def. 10), we first have to define the semantics of terms and FOL-formulas—which is done in the usual way (Def. 8)—and the semantics of programs (Def. 9).

Definition 8. *Given a state s, the valuation function val_s assigns to each term $t \in Term_{\mathbb{N}}$ a natural number $val_s(t) \in \mathbb{N}$ and to each formula one of the truth values $\underline{\mathrm{t}}$ and $\underline{\mathrm{f}}$. For terms, val_s is defined by:*

– *$val_s(x) = s(x)$ for variables $x \in Var$.*
– *$val_s(f(t_1, \dots, t_k)) = I_{\mathbb{N}}(f)(val_s(t_1), \dots, val_s(t_k))$ for $k \geq 0$.*

For FOL-formulas, val_s is defined by:

– *$val_s(true) = \underline{\mathrm{t}}$, and $val_s(false) = \underline{\mathrm{f}}$.*
– *$val_s(t_1 \doteq t_2) = \underline{\mathrm{t}}$ iff $val_s(t_1)$ and $val_s(t_2)$ are equal, and $val_s(t_1 \leq t_2) = \underline{\mathrm{t}}$ iff $val_s(t_1)$ is less than or equal to $val_s(t_2)$.*
– *$val_s(\phi \land \psi) = \underline{\mathrm{t}}$ iff $val_s(\phi) = \underline{\mathrm{t}}$ and $val_s(\psi) = \underline{\mathrm{t}}$.*
– *$val_s(\phi \lor \psi) = \underline{\mathrm{t}}$ iff $val_s(\phi) = \underline{\mathrm{t}}$ or $val_s(\psi) = \underline{\mathrm{t}}$.*
– *$val_s(\phi \to \psi) = \underline{\mathrm{t}}$ iff $val_s(\phi) = \underline{\mathrm{f}}$ or $val_s(\psi) = \underline{\mathrm{t}}$.*
– *$val_s(\forall x\, \phi) = \underline{\mathrm{t}}$ iff, for all $n \in \mathbb{N}$, $val_{s\{x \leftarrow n\}}(\phi) = \underline{\mathrm{t}}$.*

– $val_s(\exists x\, \phi) = \underline{t}$ *iff, for at least one* $n \in \mathbb{N}$, $val_{s\{x \leftarrow n\}}(\phi) = \underline{t}$.

Note, that the valuation function depends on the interpretation function of the domain of computation, which in our case is $I_{\mathbb{N}}$.

In DDL, where the modalities only refer to the final state of a program execution, the semantics of a program α is a reachability relation on states: A state s' is α-reachable from s if α terminates in s' when started in s. In DLT the situation is different. The additional modalities refer to the intermediate states as well. Since the programs are deterministic, their intermediate states form a sequence. Thus, the semantics of a program α w.r.t. a state s is the—finite or infinite—sequence of all states that α reaches when started in s, called the *trace* of α. It includes the initial state s (and the final state in case α terminates).

In the following definition, $T_1 \circ T_2$ denotes the concatenation of two traces T_1 and T_2 (the trace T_1 must be finite, T_2 may be infinite). The last element of a trace T is denoted with $last(T)$; and $|s|$ is the trace consisting of the single state s.

Definition 9. *A* trace *is a non-empty, finite or infinite sequence of states.*

Given a state s, the valuation function val_s *assigns a trace to each program* α. *It is defined by* $val_s(\alpha) = |s| \circ val'_s(\alpha)$ *where*[2]

- $val'_s(x \mathtt{:=} t) = |s\{x \leftarrow val_s(t)\}|$;
- $val'_s(\alpha\,;\beta) = val'_s(\alpha) \circ val'_{last(val_s(\alpha))}(\beta)$ *provided* $val_s(\alpha)$ *is finite, otherwise* $val'_s(\alpha\,;\beta) = val_s(\alpha)$;
- $val'_s(\mathtt{if}\ \epsilon\ \mathtt{then}\ \alpha\ \mathtt{else}\ \beta)$ *is defined to be equal to* $val'_s(\alpha)$ *if* $val_s(\epsilon) = \underline{t}$ *and to be equal to* $val'_s(\beta)$ *if* $val'_s(\epsilon) = \underline{f}$;
- $val'_s(\mathtt{while}\ \epsilon\ \mathtt{do}\ \alpha)$ *is defined as follows: Let* s_n *be the initial state of the* n*-th iteration of the loop body* α, *i.e.,* $s_1 = s$ *and, for* $n \geq 1$, $s_{n+1} = last(val_{s_n}(\alpha))$ *if* s_n *is defined and* $val_{s_n}(\alpha)$ *is finite (otherwise* s_{n+1} *remains undefined).*

  Case 1 (the loop terminates): *If for some* $n \in N$, *(i)* $val_{s_i}(\alpha)$ *is finite for all* $i \leq n$, *(ii)* $val_{s_i}(\epsilon) = \underline{t}$ *for all* $i \leq n$, *and (iii)* $val_{s_{n+1}}(\epsilon) = \underline{f}$, *then* $val'_s(\mathtt{while}\ \epsilon\ \mathtt{do}\ \alpha)$ *is the finite sequence* $val'_{s_1}(\alpha) \circ \cdots \circ val'_{s_n}(\alpha)$.

  Case 2 (each iteration terminates but the condition ϵ remains true such that the loop does not terminate): *If for all* $n \geq 1$, *(i)* $val_{s_n}(\alpha)$ *is finite and (ii)* $val_{s_n}(\epsilon) = \underline{t}$, *then* $val'_s(\mathtt{while}\ \epsilon\ \mathtt{do}\ \alpha)$ *is the infinite sequence* $val'_{s_1}(\alpha) \circ val'_{s_2}(\alpha) \circ \cdots$.

  Case 3 (some iteration does not terminate): *If for some* $n \in N$, *(i)* $val_{s_i}(\alpha)$ *is finite for* $i < n$, *(ii)* $val_{s_n}(\alpha)$ *is infinite, and (iii)* $val_{s_i}(\epsilon) = \underline{t}$ *for* $i \leq n$, *then* $val'_s(\mathtt{while}\ \epsilon\ \mathtt{do}\ \alpha)$ *is the infinite sequence* $val'_{s_1}(\alpha) \circ \cdots \circ val'_{s_n}(\alpha)$.

Now we can extend the valuation function val_s to DLT-formulas.

Definition 10. *Given a state s, the valuation function val_s assigns to each DLT-formula ϕ one of the truth values $\underline{t}$ and $\underline{f}$ as follows:*

- *If* ϕ *is an FOL-formula, then it is assigned a truth value* $val_s(\phi)$ *according to Definition 8.*
- $val_s([\alpha]\phi) = \underline{t}$ *iff* $val_s(\alpha)$ *is infinite or* $val_{s'}(\phi) = \underline{t}$ *where* $s' = last(val_s(\alpha))$.
- $val_s(\langle\alpha\rangle\phi) = \underline{t}$ *iff* $val_s(\alpha)$ *is finite and* $val_{s'}(\phi) = \underline{t}$ *where* $s' = last(val_s(\alpha))$.
- $val_s([\![\alpha]\!]\phi) = \underline{t}$ *iff* $val_{s'}(\phi) = \underline{t}$ *for all* $s' \in val_s(\alpha)$.
- $val_s(\langle\!\langle\alpha\rangle\!\rangle\phi) = \underline{t}$ *iff* $val_{s'}(\phi) = \underline{t}$ *for at least one* $s' \in val_s(\alpha)$.

Definition 11. *If* $val_s(\phi) = \underline{t}$, *then* ϕ *is said to be* true *in the state* s; *otherwise it is* false *in* s. *A formula is* valid *if it is true in all states.*

A sequent $\Gamma \vdash \Delta$ *is* valid *iff the DLT-formula* $\bigwedge \Gamma \to \bigvee \Delta$ *is valid.*

[2] Contrary to val, the valuation function val' does not include the initial state into the trace. It is only used in this definition.

Axioms	$\dfrac{}{\Gamma,\ \phi\ \vdash\ \phi,\ \Delta}$ (R1)	$\dfrac{}{\Gamma\ \vdash\ true,\ \Delta}$ (R2)	$\dfrac{}{\Gamma,\ false\ \vdash\ \Delta}$ (R3)
Rules for $\neg$	$\dfrac{\Gamma,\ \phi\ \vdash\ \Delta}{\Gamma\ \vdash\ \neg\phi,\ \Delta}$ (R4)	$\dfrac{\Gamma\ \vdash\ \phi,\ \Delta}{\Gamma,\ \neg\phi\ \vdash\ \Delta}$ (R5)	
Rules for $\wedge$	$\dfrac{\Gamma\ \vdash\ \phi,\ \Delta \quad \Gamma\ \vdash\ \psi,\ \Delta}{\Gamma\ \vdash\ \phi\wedge\psi,\ \Delta}$ (R6)	$\dfrac{\Gamma,\ \phi,\ \psi\ \vdash\ \Delta}{\Gamma,\ \phi\wedge\psi\ \vdash\ \Delta}$ (R7)	
Rules for $\vee$	$\dfrac{\Gamma\ \vdash\ \phi,\ \psi,\ \Delta}{\Gamma\ \vdash\ \phi\vee\psi,\ \Delta}$ (R8)	$\dfrac{\Gamma,\ \phi\ \vdash\ \Delta \quad \Gamma,\ \psi\ \vdash\ \Delta}{\Gamma,\ \phi\vee\psi\ \vdash\ \Delta}$ (R9)	
Rules for $\rightarrow$	$\dfrac{\Gamma,\ \phi\ \vdash\ \psi,\ \Delta}{\Gamma\ \vdash\ \phi\rightarrow\psi,\ \Delta}$ (R10)	$\dfrac{\Gamma\ \vdash\ \phi,\ \Delta \quad \Gamma,\ \psi\ \vdash\ \Delta}{\Gamma,\ \phi\rightarrow\psi\ \vdash\ \Delta}$ (R11)	
Rules for $\forall$	$\dfrac{\Gamma\ \vdash\ \phi_x^{x'},\ \Delta}{\Gamma\ \vdash\ \forall x\,\phi,\ \Delta}$ (R12) where x' is new w.r.t. ϕ, Γ, Δ	$\dfrac{\Gamma,\ \forall x\,\phi,\ \phi_x^t\ \vdash\ \Delta}{\Gamma,\ \forall x\,\phi\ \vdash\ \Delta}$ (R13) where the substitution $\{x/t\}$ is admissible w.r.t. ϕ	
Rules for $\exists$	$\dfrac{\Gamma,\ \phi_x^{x'}\ \vdash\ \Delta}{\Gamma,\ \exists x\,\phi\ \vdash\ \Delta}$ (R14) where x' is new w.r.t. ϕ, Γ, Δ	$\dfrac{\Gamma\ \vdash\ \phi_x^t,\ \exists x\,\phi,\ \Delta}{\Gamma\ \vdash\ \exists x\,\phi,\ \Delta}$ (R15) where the substitution $\{x/t\}$ is admissible w.r.t. ϕ	
Weakening	$\dfrac{\Gamma\ \vdash\ \Delta}{\Gamma\ \vdash\ \phi,\ \Delta}$ (R16)	$\dfrac{\Gamma\ \vdash\ \Delta}{\Gamma,\ \phi\ \vdash\ \Delta}$ (R17)	
Cut rule	$\dfrac{\Gamma,\ \phi\ \vdash\ \Delta \quad \Gamma\ \vdash\ \phi,\ \Delta}{\Gamma\ \vdash\ \Delta}$ (R18)		

Table 1. The elementary rules of the calculus.

4 A Sequent Calculus for DL with Trace Modalities

In this section, we present a sequent calculus for DLT, which we call $\mathcal{C}_{\mathrm{DLT}}$. It is sound and relatively complete, i.e., complete up to the handling of arithmetic (see Section 5). The set of those $\mathcal{C}_{\mathrm{DLT}}$-rules in which the additional modalities $[\![\cdot]\!]$ and $\langle\!\langle\cdot\rangle\!\rangle$ do not occur forms a sound and (relatively) complete calculus for DDL. This restriction of $\mathcal{C}_{\mathrm{DLT}}$ is similar to the DDL-calculus described in [6].

Most rules of the calculus are analytic and therefore could 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 induction rule (where a useful induction hypothesis has to be found), (c) the cut rule (where the right case distinction has to be used), and (d) the quantifier rules (where the right instantiation has to be found).

In the rule schemata, Γ, Δ denote arbitrary, possibly empty multi-sets of formulas, and ϕ, ψ denote arbitrary formulas. 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, however, 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. *The* calculus $\mathcal{C}_{\mathrm{DLT}}$ *consists of the rules* (R1) *to* (R51) *shown in Tables 1–4.*

Oracle rules	$\dfrac{}{\Gamma \vdash \Delta}$ (R19) where $\bigwedge \Gamma \to \bigvee \Delta$ is a valid arithmetical FOL-formula	$\dfrac{\Gamma_1', \Gamma_2 \vdash \Delta}{\Gamma_1, \Gamma_2 \vdash \Delta}$ (R20) where $\bigwedge \Gamma_1 \to \bigwedge \Gamma_1'$ is a valid arithmetical FOL-formula
Induction	$\dfrac{\Gamma \vdash \phi(0), \Delta \quad \Gamma, \phi(n) \vdash \phi(s(n)), \Delta}{\Gamma \vdash \forall n\, \phi(n), \Delta}$ (R21) where n does not occur in Γ, Δ	

Table 2. The rules for handling arithmetic.

A sequent is derivable *(with $\mathcal{C}_{\mathrm{DLT}}$) 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* (R1, R2, R3, R19)*.*

4.1 The Elementary Rules

The elementary rules of $\mathcal{C}_{\mathrm{DLT}}$ are shown in Table 1. The table contains rules for axioms (which have no premisses and allow to close a branch in the proof tree), rules for the propositional operators and the quantifiers, weakening rules, and the cut rule. Note, that these rules form a sound and complete calculus for FOL.

4.2 Rules for Handling Arithmetic

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 FOL-formulas in the domain of computation (note that the oracle only decides pure FOL-formulas). In the case of arithmetic, the oracle is represented by rule (R19) in Table 2. Rule (R20) is an alternative formalisation of the oracle that is often more useful.

Of course, the FOL-formulas that are valid in arithmetic are not even enumerable. Therefore, in practice, the oracle can only be approximated, and rules (R19) and (R20) must be replaced by a rule (or set of rules) for computing resp. enumerating a *subset* of all valid FOL-formulas (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 FOL-formulas that occur during the verification of actual programs.

Typically, an approximation of the computation domain oracle contains a rule for structural induction. In the case of arithmetic, that is rule (R21). This rule, however, is not only used to approximate the arithmetic oracle but is indispensable for completeness. It not only applies to FOL-formulas but also to DLT-formulas containing programs; and it is needed for handling the modalities $\langle\cdot\rangle$ and $\langle\!\langle\cdot\rangle\!\rangle$ when they contain while loops (see Section 4.3).

4.3 Rules for Modalities and Programs

The rules for the modal operators and the programs they contain are shown in Table 3. As is easy to see, they basically perform a symbolic program execution.

There is a one rule for each combination of program construct (assignment, concatenation, if-then-else, while loop) and modality ($[\cdot]$, $\langle\cdot\rangle$, $[\![\cdot]\!]$, $\langle\!\langle\cdot\rangle\!\rangle$). To keep

Assignment

$$\frac{\Gamma^{x'}_{x},\ x \doteq t^{x'}_{x} \ \vdash\ \phi,\ \Delta^{x'}_{x}}{\Gamma\ \vdash\ [x\ \texttt{:=}\ t]\phi,\ \Delta}\ (\text{R22}) \qquad \frac{\Gamma^{x'}_{x},\ x \doteq t^{x'}_{x} \ \vdash\ \phi,\ \Delta^{x'}_{x}}{\Gamma\ \vdash\ \langle x\ \texttt{:=}\ t\rangle\phi,\ \Delta}\ (\text{R23})$$

where x' is new w.r.t. t, ϕ, Γ, Δ where x' is new w.r.t. t, ϕ, Γ, Δ

$$\frac{\Gamma\ \vdash\ \phi,\ \Delta \quad \Gamma^{x'}_{x},\ x \doteq t^{x'}_{x} \ \vdash\ \phi,\ \Delta^{x'}_{x}}{\Gamma\ \vdash\ \llbracket x\ \texttt{:=}\ t\rrbracket\phi,\ \Delta}\ (\text{R24}) \qquad \frac{\Gamma^{x'}_{x},\ x \doteq t^{x'}_{x} \ \vdash\ \phi^{x'}_{x},\ \phi,\ \Delta^{x'}_{x}}{\Gamma\ \vdash\ \langle\!\langle x\ \texttt{:=}\ t\rangle\!\rangle\phi,\ \Delta}\ (\text{R25})$$

where x' is new w.r.t. t, ϕ, Γ, Δ where x' is new w.r.t. t, ϕ, Γ, Δ

Concatenation

$$\frac{\Gamma\ \vdash\ [\alpha][\beta]\phi,\ \Delta}{\Gamma\ \vdash\ [\alpha\texttt{;}\beta]\phi,\ \Delta}\ (\text{R26}) \qquad \frac{\Gamma\ \vdash\ \langle\alpha\rangle\langle\beta\rangle\phi,\ \Delta}{\Gamma\ \vdash\ \langle\alpha\texttt{;}\beta\rangle\phi,\ \Delta}\ (\text{R27})$$

$$\frac{\Gamma\ \vdash\ \llbracket\alpha\rrbracket\phi,\ \Delta \quad \Gamma\ \vdash\ [\alpha]\llbracket\beta\rrbracket\phi,\ \Delta}{\Gamma\ \vdash\ \llbracket\alpha\texttt{;}\beta\rrbracket\phi,\ \Delta}\ (\text{R28}) \qquad \frac{\Gamma\ \vdash\ \langle\!\langle\alpha\rangle\!\rangle\phi,\ \langle\alpha\rangle\langle\!\langle\beta\rangle\!\rangle\phi,\ \Delta}{\Gamma\ \vdash\ \langle\!\langle\alpha\texttt{;}\beta\rangle\!\rangle\phi,\ \Delta}\ (\text{R29})$$

If-then-else

$$\frac{\Gamma,\ \epsilon\ \vdash\ [\alpha]\phi,\ \Delta \quad \Gamma,\ \neg\epsilon\ \vdash\ [\beta]\phi,\ \Delta}{\Gamma\ \vdash\ [\texttt{if}\ \epsilon\ \texttt{then}\ \alpha\ \texttt{else}\ \beta]\phi,\ \Delta}\ (\text{R30})$$

$$\frac{\Gamma,\ \epsilon\ \vdash\ \langle\alpha\rangle\phi,\ \Delta \quad \Gamma,\ \neg\epsilon\ \vdash\ \langle\beta\rangle\phi,\ \Delta}{\Gamma\ \vdash\ \langle\texttt{if}\ \epsilon\ \texttt{then}\ \alpha\ \texttt{else}\ \beta\rangle\phi,\ \Delta}\ (\text{R31})$$

$$\frac{\Gamma,\ \epsilon\ \vdash\ \llbracket\alpha\rrbracket\phi,\ \Delta \quad \Gamma,\ \neg\epsilon\ \vdash\ \llbracket\beta\rrbracket\phi,\ \Delta}{\Gamma\ \vdash\ \llbracket\texttt{if}\ \epsilon\ \texttt{then}\ \alpha\ \texttt{else}\ \beta\rrbracket\phi,\ \Delta}\ (\text{R32})$$

$$\frac{\Gamma,\ \epsilon\ \vdash\ \langle\!\langle\alpha\rangle\!\rangle\phi,\ \Delta \quad \Gamma,\ \neg\epsilon\ \vdash\ \langle\!\langle\beta\rangle\!\rangle\phi,\ \Delta}{\Gamma\ \vdash\ \langle\!\langle\texttt{if}\ \epsilon\ \texttt{then}\ \alpha\ \texttt{else}\ \beta\rangle\!\rangle\phi,\ \Delta}\ (\text{R33})$$

While

$$\frac{\Gamma\ \vdash\ \mathit{Inv},\ \Delta \quad \mathit{Inv},\ \epsilon\ \vdash\ [\alpha]\mathit{Inv} \quad \mathit{Inv},\ \neg\epsilon\ \vdash\ \phi}{\Gamma\ \vdash\ [\texttt{while}\ \epsilon\ \texttt{do}\ \alpha]\phi,\ \Delta}\ (\text{R34})$$

where *Inv* is an arbitrary DLT-formula

$$\frac{\Gamma\ \vdash\ \epsilon,\ \Delta \quad \Gamma\ \vdash\ \langle\alpha\rangle\langle\texttt{while}\ \epsilon\ \texttt{do}\ \alpha\rangle\phi,\ \Delta}{\Gamma\ \vdash\ \langle\texttt{while}\ \epsilon\ \texttt{do}\ \alpha\rangle\phi,\ \Delta}\ (\text{R35}) \qquad \frac{\Gamma\ \vdash\ \neg\epsilon,\ \Delta \quad \Gamma\ \vdash\ \phi,\ \Delta}{\Gamma\ \vdash\ \langle\texttt{while}\ \epsilon\ \texttt{do}\ \alpha\rangle\phi,\ \Delta}\ (\text{R36})$$

$$\frac{\Gamma\ \vdash\ \mathit{Inv},\ \Delta \quad \mathit{Inv},\ \epsilon\ \vdash\ [\alpha]\mathit{Inv} \quad \mathit{Inv},\ \epsilon\ \vdash\ \llbracket\alpha\rrbracket\phi \quad \mathit{Inv},\ \neg\epsilon\ \vdash\ \phi}{\Gamma\ \vdash\ \llbracket\texttt{while}\ \epsilon\ \texttt{do}\ \alpha\rrbracket\phi,\ \Delta}\ (\text{R37})$$

where *Inv* is an arbitrary DLT-formula

$$\frac{\Gamma\ \vdash\ \epsilon,\ \Delta \quad \Gamma\ \vdash\ \langle\alpha\rangle\langle\!\langle\texttt{while}\ \epsilon\ \texttt{do}\ \alpha\rangle\!\rangle\phi,\ \Delta}{\Gamma\ \vdash\ \langle\!\langle\texttt{while}\ \epsilon\ \texttt{do}\ \alpha\rangle\!\rangle\phi,\ \Delta}\ (\text{R38})$$

$$\frac{\Gamma,\ \neg\epsilon\ \vdash\ \phi,\ \Delta \quad \Gamma,\ \epsilon\ \vdash\ \langle\!\langle\alpha\rangle\!\rangle\phi,\ \Delta}{\Gamma\ \vdash\ \langle\!\langle\texttt{while}\ \epsilon\ \texttt{do}\ \alpha\rangle\!\rangle\phi,\ \Delta}\ (\text{R39})$$

Table 3. Rules for the modal operators.

the description of our calculus compact we only give rules for the case where the modal formula is on the right side of a sequent. That is sufficient for completeness because using the cut rule (R18) and the rules for negated modalities (R48) to (R51) (see Table 4), every modal formula on the left side of a sequent can be turned into an equivalent formula on the right side of the sequent. For example, from the proof obligation $[\![\alpha]\!]\phi \vdash$ we get the proof obligation $\vdash \neg[\![\alpha]\!]\phi$ with the cut rule, which then can be turned into $\vdash \langle\!\langle\alpha\rangle\!\rangle\neg\phi$ applying rule (R50).

Rules for Assignments The rules for the modalities $[\cdot]$ (R22) and $\langle\cdot\rangle$ (R23) are the traditional assignment rules of calculi for first-order DL. They introduce a new variable x' representing the old value of x before the assignment $x := t$ is executed. In the premisses of the assignment rules, both x and x' occur because the premisses express the relation between the old and the new value of x without using an explicit assignment. Since assignments always terminate, there is no difference between the two rules.

Note that the premiss and the conclusion of these rules are not necessarily equivalent. But if one is valid then the other is valid as well.

Example 1. Consider the valid sequent $x \doteq 5 \vdash \langle x := x+1\rangle x \doteq 6$. Applying rule (R23) yields the new sequent $x' \doteq 5,\ x \doteq x'+1 \vdash x \doteq 6$. It can be read as: "If the old value of x is 5 and its new value is its old value plus 1, then the new value of x is 6." This exactly captures the meaning of the original sequent.

Assignments $x := t$ are atomic programs. By definition, their semantics is a trace consisting of the initial state s and the final state $s' = s\{x \leftarrow val_s(t)\}$. Therefore, the meaning of $[\![x := t]\!]\phi$ is that ϕ is true in both s and s', which is what the two premisses of rule (R24) express. The formula $\langle\!\langle x := t\rangle\!\rangle\phi$, on the other hand, is true (in s) if ϕ is true in at least one of the two states. Note, that the two formulas ϕ and $\phi_x^{x'}$ in the premiss of rule (R25), which express that ϕ is true in s resp. s', are implicitly disjunctively connected.

Example 2. We use rule (R24) to show that $x \doteq 5 \vdash [\![x := x+1]\!]x \leq 6$ is a valid sequent. This results in the two new proof obligations $x \doteq 5 \vdash x \leq 6$ and $x' \leq 5,\ x \doteq x'+1 \vdash x \leq 6$. They state that $x \leq 6$ is true in both the initial and the final state of the assignment.

Let $even(x)$ be an abbreviation for the FOL-formula $\exists y\,(x \doteq 2 * y)$. To prove the validity of $\vdash \langle\!\langle x := x+1\rangle\!\rangle even(x)$, we apply rule (R25) and get the new proof obligation $x \doteq x'+1 \vdash even(x),\ even(x')$, which is obviously valid.

Rules for Concatenation Again, the rules for the modalities $[\cdot]$ (R26) and $\langle\cdot\rangle$ (R27) are the traditional rules for first-order DL. They are based on the equivalences $[\alpha;\beta]\phi \leftrightarrow [\alpha][\beta]\phi$ resp. $\langle\alpha;\beta\rangle\phi \leftrightarrow \langle\alpha\rangle\langle\beta\rangle\phi$.

In the case of the $[\![\cdot]\!]$ modality, the concatenation rule (R28) branches. To show that a formula ϕ is true throughout the execution of $\alpha;\beta$, one has to prove (a) that ϕ is true throughout the execution of α, i.e. $[\![\alpha]\!]\phi$, and (b) provided α terminates, that ϕ is true throughout the execution of β that is started in the final state of α, i.e. $[\alpha][\![\beta]\!]\phi$.

The concatenation rule for $\langle\!\langle\cdot\rangle\!\rangle$ (R29) does not branch. A formula ϕ is true at least once during the execution of $\alpha;\beta$ if (a) it is true at least once during the execution of α, *or* (b) α terminates and ϕ is true at least once during the execution of β that is started in the final state of α.[3]

Rules for If-then-else The rules for if-then-else conditionals have the same form for all four modalities, and for the modalities $[\cdot]$ and $\langle\cdot\rangle$ they are the same as in calculi for standard DDL.

[3] For non-deterministic versions of DL, rule (R29) is only sound provided that the following semantics is chosen for the $\langle\!\langle\cdot\rangle\!\rangle$ modality: $\langle\!\langle\alpha\rangle\!\rangle\phi$ is true iff ϕ is true at least once in *some* of the (several) traces of α. If, however, a non-deterministic semantics is chosen where ϕ must be true at least once in *every* trace of α (as Pratt did for the propositional case [10]), then rule (R29) is *not* correct, and indeed we failed to find a sound rule for that kind of semantics.

Rules for While Loops The rules for while loops in the modalities $[\cdot]$ and $[\![\cdot]\!]$, (R34) resp. (R37), use a loop *invariant*, i.e., a DLT-formula that must be true before and after each execution of the loop body. Three premisses of (R37) are the same as the premisses of (R34). The first one expresses that the invariant *Inv* holds in the current state, i.e., before the loop is started. The second premiss expresses that *Inv* is indeed an invariant, i.e., if it holds before executing the loop body α, then it holds again if and when α terminates. And the third premiss expresses that ϕ—the formula that supposedly holds after resp. throughout executing the loop—is a logical consequence of the invariant and the negation of the loop condition ϵ, i.e., is true when the loop terminates. For the $[\![\cdot]\!]$ modality, this third premiss is only needed for the case that ϵ is false from the beginning and the loop body α is never executed. The rule for $[\![\cdot]\!]$ (R37) has an additional fourth premiss, which requires to show that ϕ remains true throughout the execution of α if the invariant is true at the beginning (this latter condition follows from the other premisses).

Example 3. Let α be the loop `while` *true* `do` x `:=` 0. Then, because α does *not* terminate, the sequent $x \doteq 0 \vdash [\![\alpha;x := 1]\!]x \doteq 0$ is valid. To prove that, we apply rule (R28), which results in the two new proof obligations $x \doteq 0 \vdash [\![\alpha]\!]x \doteq 0$ and $x \doteq 0 \vdash [\alpha][\![x := 1]\!]x \doteq 0$. Both are easy to derive with the rules for while loops, namely the former one with rule (R37) and the invariant $x \doteq 0$ and the latter one with rule (R34) and the invariant *true*.

The modalities $\langle\cdot\rangle$ and $\langle\!\langle\cdot\rangle\!\rangle$ are handled in a different way. Two rules are provided for each of them. One rule, (R35) resp. (R38), allows to "unwind" the loop, i.e., to symbolically execute it once, provided that the loop condition ϵ is true in the current state. The other rule, (R36) resp. (R39), is used if "unwinding" the loop is not useful. For the $\langle\cdot\rangle$ modality that is the case if ϵ is false and the loop terminates immediately. Rule (R39) for the $\langle\!\langle\cdot\rangle\!\rangle$ modality applies in case the formula ϕ—which supposedly is true at least once during the execution of the loop—becomes true before or during the first execution of the loop body.

The rules for $\langle\cdot\rangle$ and $\langle\!\langle\cdot\rangle\!\rangle$ only work in combination with the induction rule, as the following example demonstrates.

Example 4. Consider the sequent $x \doteq 0 \vdash \langle\!\langle \texttt{while } \mathit{true} \texttt{ do } x := x + 1\rangle\!\rangle x \doteq k$. It states that, if the value of x is 0 initially, then during the execution of the non-terminating loop, x will at least once have the value k.

To show that this sequent is valid, we first use the induction rule to prove that $\vdash \forall n\, \phi(n)$ is valid, where

$$\phi(n) \;=\; (x \leq k \land n + x \doteq k) \to \langle\!\langle \texttt{while } \mathit{true} \texttt{ do } x := x + 1\rangle\!\rangle x \doteq k \;\;,$$

from which then the original proof obligation can be derived instantiating n with k. The first premiss of the induction rule, $\vdash \phi(0)$, can easily be derived with rule (R39) as $x \doteq k$ is immediately true in case $n = 0$. The second premiss, $\phi(n) \vdash \phi(n + 1)$, can be derived by first applying the cut rule to distinguish the cases $x < k$ and $x \doteq k$. In the first case, the unwind rule (R38) can be used successfully; and the second case is again easily covered with rule (R39).

4.4 Miscellaneous Other Rules

There are three types of miscellaneous other rules (see Table 4). The first type are the generalisation rules (R40) to (R43), which allow to derive $\mathit{Op}\,\phi \vdash \mathit{Op}\,\psi$ from $\phi \vdash \psi$ where *Op* is any of the four modal operators.

Second, there are rules, (R44) to (R47), that allow to replace (universal) quantifications by modalities. They are similar to the quantifier instantiation

Generalisation

$$\frac{\phi \vdash \psi}{[\alpha]\phi \vdash [\alpha]\psi}\ \text{(R40)} \qquad \frac{\phi \vdash \psi}{\langle\alpha\rangle\phi \vdash \langle\alpha\rangle\psi}\ \text{(R41)}$$

$$\frac{\phi \vdash \psi}{[\![\alpha]\!]\phi \vdash [\![\alpha]\!]\psi}\ \text{(R42)} \qquad \frac{\phi \vdash \psi}{\langle\!\langle\alpha\rangle\!\rangle\phi \vdash \langle\!\langle\alpha\rangle\!\rangle\psi}\ \text{(R43)}$$

Quantifier/modality rules

$$\frac{\Gamma,\ \forall x_1 \ldots \forall x_k\, \phi,\ [\alpha]\phi \vdash \Delta}{\Gamma,\ \forall x_1 \ldots \forall x_k\, \phi \vdash \Delta}\ \text{(R44)} \qquad \frac{\Gamma \vdash \langle\alpha\rangle\phi,\ \exists x_1 \ldots \exists x_k\, \phi,\ \Delta}{\Gamma \vdash \exists x_1 \ldots \exists x_k\, \phi,\ \Delta}\ \text{(R45)}$$

where $Var(\alpha) \subseteq \{x_1, \ldots, x_k\}$ where $Var(\alpha) \subseteq \{x_1, \ldots, x_k\}$

$$\frac{\Gamma,\ \forall x_1 \ldots \forall x_k\, \phi,\ [\![\alpha]\!]\phi \vdash \Delta}{\Gamma,\ \forall x_1 \ldots \forall x_k\, \phi \vdash \Delta}\ \text{(R46)} \qquad \frac{\Gamma \vdash \langle\!\langle\alpha\rangle\!\rangle\phi,\ \exists x_1 \ldots \exists x_k\, \phi,\ \Delta}{\Gamma \vdash \exists x_1 \ldots \exists x_k\, \phi,\ \Delta}\ \text{(R47)}$$

where $Var(\alpha) \subseteq \{x_1, \ldots, x_k\}$ where $Var(\alpha) \subseteq \{x_1, \ldots, x_k\}$

Rules for negated modalities

$$\frac{\Gamma \vdash \langle\alpha\rangle\neg\phi,\ \Delta}{\Gamma \vdash \neg[\alpha]\phi,\ \Delta}\ \text{(R48)} \qquad \frac{\Gamma \vdash [\alpha]\neg\phi,\ \Delta}{\Gamma \vdash \neg\langle\alpha\rangle\phi,\ \Delta}\ \text{(R49)}$$

$$\frac{\Gamma \vdash \langle\!\langle\alpha\rangle\!\rangle\neg\phi,\ \Delta}{\Gamma \vdash \neg[\![\alpha]\!]\phi,\ \Delta}\ \text{(R50)} \qquad \frac{\Gamma \vdash [\![\alpha]\!]\neg\phi,\ \Delta}{\Gamma \vdash \neg\langle\!\langle\alpha\rangle\!\rangle\phi,\ \Delta}\ \text{(R51)}$$

Table 4. Miscellaneous rules.

rules (R13) and (R15) and are based on the fact that, for example, $[\![\alpha(x)]\!]\phi$ is true in a state s if $\forall x\, \phi$ is true in s and x is the only variable in $\alpha(x)$.

And third, there are rules, (R48) to (R51), implementing the equivalences $\neg[\alpha]\phi \leftrightarrow \langle\alpha\rangle\neg\phi$ and $\neg[\![\alpha]\!]\phi \leftrightarrow \langle\!\langle\alpha\rangle\!\rangle\neg\phi$.

5 Soundness and Relative Completeness

5.1 Soundness

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

Theorem 1. *For all rules schemata of the calculus $\mathcal{C}_{\mathrm{DLT}}$,* (R1) *to* (R51)*, the following holds: If all premisses of a rule schema instance are valid sequents, then its conclusion is a valid sequent.*

Proving the above theorem is not difficult. The proof is, however, quite large as soundness has to be shown separately vor each rule. For the assignment rules, the proof is based on a substitution lemma and is technically involved.

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

5.2 Relative Completeness

The calculus $\mathcal{C}_{\mathrm{DLT}}$ 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, (R19) and (R20). If the domain is extended with other data types, $\mathcal{C}_{\text{DLT}}$ 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* $\mathcal{C}_{\text{DLT}}$*.*

Corollary 2. *If* ϕ *is a valid DLT-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 (it can be found in [12]). The proof technique is the same as that used by Harel [4] to prove relative completeness of his sequent calculus for first-order DL. The following lemmata are central to the completeness proof.

Lemma 1. *For every DLT-formula* F_{DLT} *there is an (arithmetical) FOL-formula* F_{FOL} *that is equivalent to* F_{DLT}*, i.e.,* $val_s(F_{\text{DLT}}) = val_s(F_{\text{FOL}})$ *for all states* s*.*

The above lemma states that DLT 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 Gödelisation, arithmetic allows to encode program states (to be more precise, the values of all the variables occurring in a program) and (finite) traces into a single number. Note that the lemma states a property of the logic DLT that is independent of the calculus.

Lemma 1 implies that a DLT-formula F_{DLT} could be decided by constructing an equivalent FOL-formula F_{FOL} 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 formula F_{FOL} would be too complex to prove automatically or interactively). And, indeed, the calculus $\mathcal{C}_{\text{DLT}}$ does no work that way.

It may be surprising that the (relative) completeness of $\mathcal{C}_{\text{DLT}}$ requires an expressive computation domain and is lost if a simpler domain and less expressive data structures are used. The reason is that a simpler domain may not allow to express the required invariants resp. induction hypotheses to handle while loops.

Lemma 2. *Let* ϕ *and* ψ *be FOL-formulas, let* α *be a program, and let* M_α *be any of the modalities* $[\alpha], \langle\alpha\rangle, [\![\alpha]\!], \langle\!\langle\alpha\rangle\!\rangle$*.*
If the sequent $\phi \vdash M_\alpha \psi$ *is valid, then it is derivable with* $\mathcal{C}_{\text{DLT}}$*.*

Lemma 2 is at the core of the completeness of $\mathcal{C}_{\text{DLT}}$. It is proven by induction on the complexity of the program α, and the proof would not go through if the calculus would lack important rules.[4]

Besides Lemmata 1 and 2, the completeness proof makes use of the fact that the calculus has the necessary rules (a) for the operators of classical logic (in particular all propositional tautologies can be derived), and (b) for generalisation, (R40) to (R43).

6 Extended Example

Consider the following program:

```
while $true$ do
  if $y \doteq 1$ then
     $x$ := $x + 1$; if $x \doteq 2$ then $y$ := 0 else $y$ := 1      $\} =: \alpha$
  else
     $x$ := 0; $y$ := 1      $\} =: \beta$
```

[4] Not all rules are indispensable. Some can be derived from other rules; they are included for convenience.

It consists of a non-terminating while loop. The loop body changes the value of x between 0 and 2 and the value of y between 0 and 1. We want to prove that $0 \leq x \leq 2$ is true in all states reached by this program, if it is started in a state where $val_s(x) = 0$ and $val_s(y) = 1$.[5] The complete proof is shown in Figure 1. Its initial proof obligation is the sequent

$$x \doteq 0,\ y \doteq 1 \vdash [\![\texttt{while } \mathit{true} \texttt{ do if } y \doteq 1 \texttt{ then } \alpha \texttt{ else } \beta]\!] 0 \leq x \leq 2 \tag{1}$$

First, the while loop is eliminated applying rule (R37) with the invariant

$$\mathit{Inv} \quad := \quad 0 \leq y \leq 1 \ \wedge\ (y \doteq 0 \rightarrow x \doteq 1 \vee x \doteq 2) \ \wedge\ (y \doteq 1 \rightarrow x \doteq 0) \ .$$

The formula $0 \leq x \leq 2$, which is a logical consequence of Inv, does not describe the behaviour of the loop in sufficient detail and, therefore, is not a suitable invariant itself. The result of applying rule (R37) to (1) are the following four new proof obligations:

$$x \doteq 0,\ y \doteq 1 \vdash \mathit{Inv} \tag{2}$$
$$\mathit{Inv},\ \mathit{true} \vdash [\texttt{if } y \doteq 1 \texttt{ then } \alpha \texttt{ else } \beta]\mathit{Inv} \tag{3}$$
$$\mathit{Inv},\ \mathit{true} \vdash [\![\texttt{if } y \doteq 1 \texttt{ then } \alpha \texttt{ else } \beta]\!] 0 \leq x \leq 2 \tag{4}$$
$$\mathit{Inv},\ \neg\mathit{true} \vdash 0 \leq x \leq 2. \tag{5}$$

Proof obligation (2) can immediately be derived with rule (R19). And, applying rule (R5) to (5) yields a sequent (5′) with *true* on the right, which can be derived with rule (R2).

In the sequel, we concentrate on the proof of (4). Proof obligation (3) can be derived in a similar way as (4); its derivation is omitted due to lack of space.

The next step is the application of rule (R32) to (4) to symbolically execute the if-then-else statement. The result are the following two proof obligations.

$$\mathit{Inv},\ \mathit{true},\ y \doteq 1 \vdash [\![x := x + 1\texttt{; if } x \doteq 2 \texttt{ then } y := 0 \texttt{ else } y := 1]\!] 0 \leq x \leq 2 \tag{6}$$
$$\mathit{Inv},\ \mathit{true},\ \neg y \doteq 1 \vdash [\![x := 0\texttt{; } y := 1]\!] 0 \leq x \leq 2 \tag{7}$$

Eliminating the concatenations in (6) and (7) with applications of rule (R28) yields (8) and (9) resp. (10) and (11).

$$\mathit{Inv},\ \mathit{true},\ y \doteq 1 \vdash [\![x := x + 1]\!] 0 \leq x \leq 2 \tag{8}$$
$$\mathit{Inv},\ \mathit{true},\ y \doteq 1 \vdash [x := x + 1][\![\texttt{if } x \doteq 2 \texttt{ then } y := 0 \texttt{ else } y := 1]\!] 0 \leq x \leq 2 \tag{9}$$
$$\mathit{Inv},\ \mathit{true},\ \neg y \doteq 1 \vdash [\![x := 0]\!] 0 \leq x \leq 2 \tag{10}$$
$$\mathit{Inv},\ \mathit{true},\ \neg y \doteq 1 \vdash [x := 0][\![y := 1]\!] 0 \leq x \leq 2. \tag{11}$$

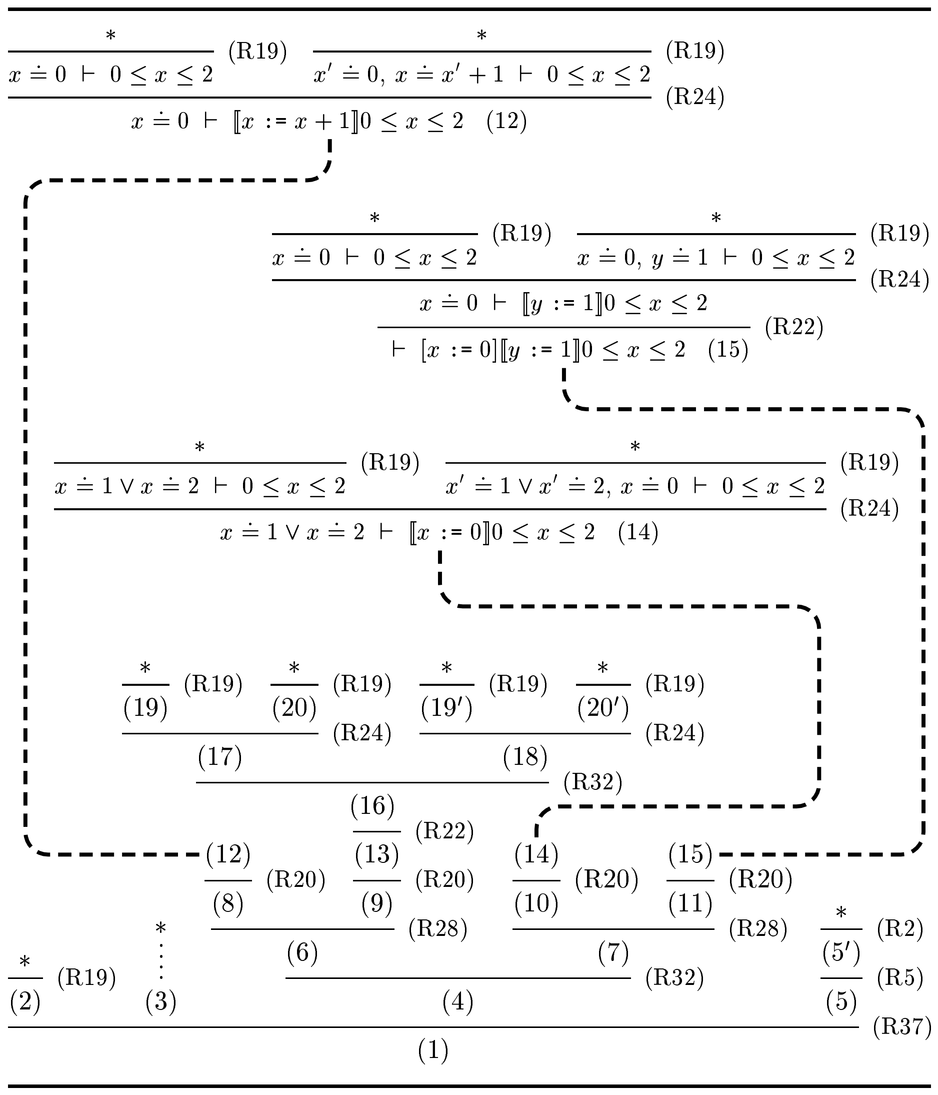
Next, we simplify (and weaken) the left sides of (8)–(11) with the arithmetic rule (R20) (this is not really necessary but the sequents get shorter and easier to understand). The result are the following sequents, respectively:

$$x \doteq 0 \vdash [\![x := x + 1]\!] 0 \leq x \leq 2 \tag{12}$$
$$x \doteq 0 \vdash [x := x + 1][\![\texttt{if } x \doteq 2 \texttt{ then } y := 0 \texttt{ else } y := 1]\!] 0 \leq x \leq 2 \tag{13}$$
$$x \doteq 1 \vee x \doteq 2 \vdash [\![x := 0]\!] 0 \leq x \leq 2 \tag{14}$$
$$\vdash [x := 0][\![y := 1]\!] 0 \leq x \leq 2 \tag{15}$$

The derivations of proof obligations (12), (14), and (15) need no further explanation and are shown in Figure 1. To derive proof obligation (13), we apply rule (R22) and get

$$x' \doteq 0,\ x \doteq x' + 1 \vdash [\![\texttt{if } x \doteq 2 \texttt{ then } y := 0 \texttt{ else } y := 1]\!] 0 \leq x \leq 2 \tag{16}$$

[5] In this section, we use $0 \leq x \leq 2$ as an abbreviation for $0 \leq x \wedge x \leq 2$.



(R19)
(R24)
$x \doteq 0 \vdash 0 \leq x \leq 2$
$x' \doteq 0,\ x \doteq x' + 1 \vdash 0 \leq x \leq 2$
$x \doteq 0 \vdash [\![x := x + 1]\!] 0 \leq x \leq 2$ (12)
$x \doteq 0,\ y \doteq 1 \vdash 0 \leq x \leq 2$
$x \doteq 0 \vdash [\![y := 1]\!] 0 \leq x \leq 2$
(R22)
$\vdash [x := 0][\![y := 1]\!] 0 \leq x \leq 2$ (15)
$x \doteq 1 \vee x \doteq 2 \vdash 0 \leq x \leq 2$
$x' \doteq 1 \vee x' \doteq 2,\ x \doteq 0 \vdash 0 \leq x \leq 2$
$x \doteq 1 \vee x \doteq 2 \vdash [\![x := 0]\!] 0 \leq x \leq 2$ (14)
(19) (20) (19′) (20′) (17) (18) (R32) (16) (13) (12) (14) (15) (R20) (8) (9) (10) (11) (R28) (6) (7) (4) (2) (3) (5′) (5) (R2) (R5) (R37) (1)

Fig. 1. The derivation described in Section 6.

The if-then-else statement is symbolically executed with rule (R32), and we get

$$x' \doteq 0,\ x \doteq x' + 1,\ x \doteq 2 \vdash [\![y := 0]\!] 0 \leq x \leq 2 \qquad (17)$$

$$x' \doteq 0,\ x \doteq x' + 1,\ \neg x \doteq 2 \vdash [\![y := 1]\!] 0 \leq x \leq 2 \qquad (18)$$

Proof obligation (17) is derived by applying rule (R24), which yields:

$$x' \doteq 0,\ x \doteq x' + 1,\ x \doteq 2 \vdash 0 \leq x \leq 2 \qquad (19)$$

$$x' \doteq 0,\ x \doteq x' + 1,\ x \doteq 2,\ y \doteq 0 \vdash 0 \leq x \leq 2 \qquad (20)$$

It is easy to check that (19) and (20) are valid FOL-sequents and can therefore be derived with the oracle rule for arithmetic (R19).

Applying rule (R24) to (18) yields similar FOL-sequents (19′) and (20′), which differ from (19) and (20) in that they contain $\neg x \doteq 2$ instead of $x \doteq 2$ and $y \doteq 1$ instead of $y \doteq 0$. They, too, can be derived with the oracle (R19).

7 Future Work

Future work includes an implementation of our calculus $\mathcal{C}_{\mathrm{DLT}}$, which would allow to carry out case studies going beyond the simple examples shown in this paper and to test the usefulness of DLT in practice.

A useful extension of $\mathcal{C}_{\mathrm{DLT}}$ for practical applications may be special rules for formulas of the form $[\alpha]\phi \wedge [\![\alpha]\!]\psi$, such that splitting the two conjuncts is avoided and they do not have to be handled in separate—but similar—sub-proofs.

Also, it may be useful to consider (a) a non-deterministic version of DLT, and (b) extensions of DLT with further modalities such as "α preserves ϕ", which expresses that, once ϕ becomes true in the trace of α, it remains true throughout the rest of the trace. It seems, however, to be difficult to give a (relatively) complete calculus for this modality.

Acknowledgements

We thank W. Ahrendt, E. Habermalz, W. Menzel, and P. H. Schmitt for fruitful discussions and comments on earlier versions of this paper.

References

1. W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, and P. H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In M. Ojeda-Aciego, I. P. de Guzman, G. Brewka, and L. M. Pereira, editors, *Proceedings, Logics in Artificial Intelligence (JELIA), Malaga, Spain*, LNCS 1919. Springer, 2000.
2. K. R. Apt. Ten years of Hoare logic: A survey – part I. *ACM Transactions on Programming Languages and Systems*, 1981.
3. B. Beckert. A Dynamic Logic for Java Card. In *Proceedings, 2nd ECOOP Workshop on Formal Techniques for Java Programs, Cannes, France*, pages 111–119, 2000. Also presented at *Java Card Workshop (JCW), Cannes, France, 2000*, and submitted to the proceedings of *JCW* to be published in Springer LNCS.
4. D. Harel. *First-order Dynamic Logic*. LNCS 68. Springer, 1979.
5. D. Harel. Dynamic Logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Volume II: Extensions of Classical Logic*, pages 497–604. Reidel, 1984.
6. M. Heisel, W. Reif, and W. Stephan. A Dynamic Logic for program verification. In A. Meyer and M. Taitslin, editors, *Proceedings, Logic at Botic, Pereslavl-Zalessky, Russia*, LNCS 363. Springer, 1989.
7. D. Hutter, B. Langenstein, C. Sengler, J. H. Siekmann, and W. Stephan. Deduction in the Verification Support Environment (VSE). In M.-C. Gaudel and J. Woodcock, editors, *Proceedings, International Sympoium of Formal Methods Europe (FME), Oxford, UK*, LNCS 1051. Springer, 1996.
8. D. Kozen and J. Tiuryn. Logic of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 14, pages 89–133. Elsevier, 1990.
9. V. R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proceedings, 18th IEEE Symposium on Foundation of Computer Science*, pages 109–121, 1977.
10. V. R. Pratt. Process logic: Preliminary report. In *Proceedings, ACM Symposium on Principles of Programming Languages (POPL), San Antonio/TX, USA*, 1979.
11. W. Reif. The KIV-approach to software verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, LNCS 1009. Springer, 1995.
12. S. Schlager. Erweiterung der Dynamischen Logik um temporallogische Operatoren. Studienarbeit, Fakultät für Informatik, Universität Karlsruhe, 2000. In German. Available at: `ftp://i12ftp.ira.uka.de/pub/beckert/schlager.ps.gz`.