CARL VON OSSIETZKY universität OLDENBURG
FAKULTÄT II
DEPARTMENT FÜR INFORMATIK

A Theorem Prover for Differential Dynamic Logic

Deductive Verification of Hybrid Systems

April 17, 2007

Jan-David Quesel

Carl von Ossietzky Universität Oldenburg
Fakultät II
Department für Informatik
Abteilung Entwicklung korrekter Systeme

Gutachter: Prof. Dr. Ernst-Rüdiger Olderog
Dipl.-Inform. André Platzer

Abstract

This thesis aims at the computer aided verification of hybrid systems using deductive techniques. We have developed an interactive verification tool on the basis of a sound sequent calculus for dℒ. The logic dℒ is a dynamic logic with a special focus on the specification and verification of hybrid systems. Our implementation extends the theorem prover component of the KeY system with rules and data structures for handling dℒ formulas. Additionally, we have integrated KeY with the computer algebra system Mathematica to handle quantifiers over the reals and real arithmetic. In order to demonstrate that our implementation can be used for verifying larger systems, we prove safety in a case study from the context of the European Train Control System (ETCS).

Acknowledgements

At this point I like to thank those who supported me during the work on this thesis. First of all I want to express my great gratefulness to André Platzer, for the various discussions, the useful hints and for giving me the opportunity to write this thesis by recommending the subject. Also I like to thank Prof. Dr. Ernst-Rüdiger Olderog for pointing out the most relevant aspects of the subject. From the KeY team I like to thank Philipp Ruemmer, Steffen Schlager, Benjamin Weiss, Daniel Larsson and most of all Richard Bubel for their advice on the topic of the design and the implementation of the KeY prover and dynamic logic in general. Some people have spent time on reading preliminary versions of this thesis as well. These are by name Dominik Denker, Andreas Schäfer, Sven Linker and my father Carsten Quesel. Last but not least, I would like to thank Christin Boldt for her patience and support.

Contents

1 Introduction **1**
1.1 Overview . . . 1
1.2 Basic Definitions . . . 2
1.3 Propositional Logic . . . 2
1.3.1 Syntax . . . 2
1.3.2 Semantics . . . 3
1.4 First-Order Logic . . . 3
1.4.1 Syntax . . . 3
1.4.2 Semantics . . . 4
1.5 KeY . . . 5

2 Differential Dynamic Logic **7**
2.1 Overview . . . 7
2.2 Syntax . . . 7
2.3 Semantics . . . 9
2.4 Sequent Calculus for $\mathsf{d}\mathcal{L}$ . . . 13
2.4.1 Sequent Calculi . . . 13
2.4.2 Rules for Propositional Logic . . . 15
2.4.3 Rules for FOL . . . 15
2.4.4 Rules for Hybrid Programs . . . 20
2.4.5 Soundness Proofs . . . 23

3 Design **37**
3.1 Overview . . . 37
3.1.1 Architecture of the KeY Prover . . . 37
3.1.2 Calculus Embedding . . . 39
3.2 Integration . . . 43
3.2.1 JAVA Embedding of Hybrid Programs . . . 43
3.2.2 Abstract Syntax of $\mathsf{d}\mathcal{L}$ Formulas in KeY . . . 44
3.2.3 Syntax Tree . . . 46
3.2.4 Data Structures for Hybrid Programs . . . 47
3.3 Mathematica Integration . . . 51

4 Implementation **57**
4.1 Overview . . . 57
4.2 KeY Extension . . . 57
4.2.1 Parsing . . . 57
4.2.2 Calculus Embedding . . . 59

4.2.3 Integration Challenges . . . 71
4.2.4 Strategy . . . 71
4.3 Mathematica Integration . . . 75
4.4 Usage . . . 76
4.4.1 Input Format . . . 76
4.4.2 Tool Overview . . . 77

5 Case Study **83**
5.1 Overview . . . 83
5.2 Formal Model . . . 85
5.3 Verification . . . 90
5.3.1 RBC Behavior . . . 91
5.3.2 Train Controller . . . 93

6 Related Work **97**

7 Conclusions **101**

Appendices **103**

A Design Patterns **105**
A.1 Architectural Patterns . . . 105
A.2 Fundamental Patterns . . . 105
A.2.1 Delegation . . . 105
A.2.2 Immutable Object . . . 105
A.2.3 Marker Interface Pattern . . . 106
A.3 Creational Patterns . . . 106
A.3.1 Abstract Factory . . . 106
A.3.2 Lazy Initialization . . . 106
A.3.3 Singleton . . . 106
A.4 Behavioral Pattern . . . 108
A.4.1 Visitor . . . 108
A.4.2 Iterator Pattern . . . 108

List of Figures

2.1 Propositional rules 16
2.2 Example proof for the quantifier handling 17
2.3 First-order rules 18
2.4 Side deduction 19
2.5 Alternative rule for existential quantifier on the right side 20
2.6 Proof structure with alternative rule for existential quantifier on the right side 20
2.7 Rules for modalities 24

3.1 KeY Architecture 38
3.2 Example for a taclet 40
3.3 Relation between sequent calculus rules and taclets 41
3.4 Example for a taclet using a metaoperator 42
3.5 JAVA embedded differential equation system 44
3.6 Data structures for dℒ programs resulting from productions α and ATOMIC-α 48
3.7 Data structures for formulas within dℒ programs generated by the productions α-FORM and α-ATOM 50
3.8 Data structures for expressions within dℒ programs generated by α-EXPR, DIFFEQUATION and DIFFEXPR 52
3.9 Data structures for functions within dℒ programs 53
3.10 Mathematica interface design 55

4.1 Two stage parsing of hybrid programs 58
4.2 Implementation of data structures for non-terminal dℒ programs . 60
4.3 Implementation of compound formulas 60
4.4 Implementation of data structures for terminal dℒ programs . . . 61
4.5 Implementation of data structures for comparisons 61
4.6 Implementation of data structures for functions 62
4.7 Metaoperators used in the taclets for dℒ 66
4.8 Taclet representation of the invariant rule R29 67
4.9 The built in rules used for the dℒ calculus 72
4.10 Rules for normalization of inequalities 74
4.11 Snapshot of the KeY Prover 79
4.12 Snapshot of the KeY Prover with context menu opened on a subformula 80
4.13 Snapshot of the KeY Prover with context menu opened on a complete formula 81

4.14 Snapshot of the Abort Program 82

5.1 Communication between the train and the RBC 83
5.2 Train blocking 3 segments 84
5.3 Train model as automata 86

A.1 Abstract class diagram for the abstract factory pattern 107
A.2 Abstract class diagram for the singleton pattern with lazy initialization 107
A.3 Abstract class diagram for the visitor pattern 108

List of Tables

3.1 Abstract syntax of dℒ formulas 45

4.1 Relation between rule classes and implementation method 60
4.2 Mapping from abstract syntax to input syntax 78

5.1 System specification 88

Chapter 1

Introduction

1.1 Overview

As safety critical systems grow in complexity, formal verification becomes more important. In this thesis we present a computer aided deductive approach to the verification of hybrid systems. We want to develop a computer aided approach, because proving safety of large systems by hand is time intensive and many proof steps can be performed automatically by the theorem prover. Additionally, when performing large proofs by hand, cases might be forgotten. The implementation supports the user at this point and keeps track of open proof goals. A deductive approach was chosen to support parameterized systems.

For this purpose we integrated the theorem prover component of the KeY system [BHS07] with Mathematica [Wol03]. Among other tools Mathematica is a commercial computer algebra system (CAS). It is known for its very powerful symbolic calculation engine. In recent versions, features for numerical calculations have been added as well.

Hybrid systems are systems with continuous and discrete state transitions. The continuous state transitions are evolutions of continuous variables along differential equations. A common example for a safety critical hybrid system is an airplane. As flight dynamics are very complex and also important for verification tasks a mathematical model is needed that does not abstract from those dynamics. Hybrid systems provide such a model. For the formal specification we use a dynamic logic [HKT00], which has a special focus on the specification and verification of hybrid systems [Hen96]. A dynamic logic is a special multi-modal logic where the modalities have a program structure. Dynamic logics are usually used for program verification in the discrete case.

The logic we use for the description of hybrid systems is called differential dynamic logic ($\mathsf{d}\mathcal{L}$) [Pla07c]. The differential dynamic logic is an extended dynamic logic where it is possible to describe continuous evolutions using differential equations. The programs within the modalities in $\mathsf{d}\mathcal{L}$ are so-called *hybrid programs*. In addition to discrete mode switching, hybrid programs can describe continuous evolutions. Operators for conditional execution, sequential composition, non-deterministic choice and non-deterministic repetition are also available.

Structure The structure of this thesis is as follows. In this chapter we will elaborate some basic definitions as well as provide definitions for propositional

and first-order logic as these terms are frequently used to identify subsets of $\mathsf{d}\mathcal{L}$. Additionally, we will provide a short description of the KeY tool.

In chapter 2 we will provide the definition of the differential dynamic logic $\mathsf{d}\mathcal{L}$, as well as a sequent calculus that can be used for verifying systems described in $\mathsf{d}\mathcal{L}$.

Chapter 3 and 4 describe our extensions to KeY and the integration with Mathematica. We will describe how we designed the extensions and how they are implemented.

In chapter 5, a case study from the context of the European Train Control System (ETCS) [ERT02] will be presented to illustrate how the extended version of KeY can be used to verify greater systems.

The last two chapters are used to compare the results of this thesis to related work, to recapitulate major findings and to point out some perspectives of future work.

1.2 Basic Definitions

In this section we introduce basic definitions for the notations used in this thesis. First we define a notation for the set of real numbers and the set $\{true, false\}$.

Definition 1. *We define the following abbreviations:*

1. *The set of real numbers is called* $\mathbb{R}$.
2. *The set of the boolean constants* $\{true, false\}$ *is called* $\mathbb{B}$.

We also need the composition of functions, thus we define it here.

Definition 2 (Composition of functions)**.** *The* composition *of two functions* $f : X \mapsto Y$ *and* $g : Y \mapsto Z$*, where* X*,* Y *and* Z *are arbitrary sets, is written as* $f \circ g$*. The semantics of this is applying the function* f *to the argument and afterwards the function* g *to the result of* f*.*

1.3 Propositional Logic

Propositional logic [Fit96] is a classic logic introduced by philosophers to formalize propositions.

1.3.1 Syntax

Let Ξ be a set of boolean constants.

- $true$ and $false$ are propositional formulas.
- If X is in Ξ, then X is a propositional formula.
- If φ and ψ are propositional formulas, then $\varphi \wedge \psi$ is a propositional formula.

- If φ is a propositional formula then $\neg\varphi$ is a propositional formula.

With these definitions we can add abbreviations for the other classical boolean combinations.

- $\varphi \vee \psi$ can be defined as $\neg(\neg\varphi \wedge \neg\psi)$
- $\varphi \rightarrow \psi$ can be defined as $\neg\varphi \vee \psi$
- $\varphi \leftrightarrow \psi$ can be defined as $(\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$

1.3.2 Semantics

The semantics of propositional formulas is given by the valuation function $val_\beta(\varphi)$ where β is the interpretation of the boolean constants. $val_\beta(\varphi)$ is defined as:

- $val_\beta(true) :=$ true
- $val_\beta(false) :=$ false
- $val_\beta(x) := \beta(x)$
- $val_\beta(\varphi \wedge \psi) := val_\beta(\varphi)$ and $val_\beta(\psi)$
- $val_\beta(\neg\varphi) :=$ not $val_\beta(\varphi)$

1.4 First-Order Logic

First-Order Logic (FOL) [Fit96] is an extension of the propositional logic introducing functions, predicates and quantifiers. The following definitions are taken from [Old02].

1.4.1 Syntax

The syntax of FOL consists of a countable infinite set of variables Var, the common logic junctors $(\neg, \wedge, \vee, \rightarrow, \leftrightarrow)$ as well as quantifiers $(\forall, \exists)$. Additionally, there are non-logic symbols supplied by the signature.

Definition 3 (Signature for FOL)**.** *A signature is a pair of two sets, $Func$ and $Pred$. The first contains function symbols and the latter contains predicate symbols. The signature also supplies the arity of these symbols.*

For the further definitions we assume Var, $Func$ and $Pred$ to be pairwise disjoint. Let $S = (Func, Pred)$ be a signature.

Definition 4 (Terms)**.** *The set of terms ($Term_S$) over a given signature S is the smallest set such that:*

- *If $X \in Var$ then $X \in Term_S$.*

- *If* $t_1, \ldots, t_n \in Term_S$, $f \in Func$ *and* f *has the arity* n *then* $f(t_1, \ldots, t_n) \in Term_S$.

Definition 5 (Formulas)**.** *The set of* formulas $Form_S$ *over a given signature* S *is the smallest set such that:*

- *If* $t_1, \ldots, t_n \in Term_S$, $p \in Pred$ *and* p *has the arity* n *then* $p(t_1, \ldots, t_n) \in Form_S$.
- *If* $F \in Form_S$ *then* $\neg F \in Form_S$.
- *If* $F, G \in Form_S$ *then* $(F \wedge G), (F \vee G), (F \rightarrow G), (F \leftrightarrow G) \in Form_S$.
- *If* $X \in Var$ *and* $F \in Form_S$ *then* $\forall XF, \exists XF \in Form_S$.

Definition 6 (Syntax)**.** *The syntax of FOL for a given signature* S *is the set:*

$$L_S = Term_S \cup Form_S$$

1.4.2 Semantics

Definition 7 (Structure)**.** *A structure* $\mathcal{M}$ *for the signature is a pair of two sets* $(D_\mathcal{M}, \jmath_\mathcal{M})$. *The first is non-empty and is called the universe. The latter is the interpretation.*

Definition 8 (Interpretation)**.** *The interpretation* $\jmath$ *is a relation that maps every function symbol* $f \in Func$ *to a function with arity* n*:*

$$\jmath_\mathcal{M}(f) : D_\mathcal{M}^n \rightarrow D_\mathcal{M}$$

The same holds for predicate symbols, but the range of those is boolean.

$$\jmath_\mathcal{M}(p) : D_\mathcal{M}^n \rightarrow \mathbb{B}$$

Definition 9 (Valuation)**.** *A valuation* β *is a function that maps every variable to an element in the universe.*

$$\beta : Var \rightarrow D_\mathcal{M}$$

Definition 10 (Semantic modification of valuations)**.** *A* semantic modification of a valuation β *is written as* $\beta[X \mapsto d]$ *which is a valuation identical to* β *expect for the valuation of* X*, which is* $d \in D_\mathcal{M}$.

Definition 11 (Semantics of terms)**.** *The semantics of a term* $t \in Term_S$ *depending on an interpretation* $\jmath$ *and a valuation of variables* β *is given by a valuation function*

$$val_{\jmath,\beta}(t) : (Var \times D_\mathcal{M}) \rightarrow D_\mathcal{M}$$

This function is inductivly defined by

$$\begin{aligned} val_{\jmath,\beta}(X) &= \beta(X) \text{ iff } X \in Var \\ val_{\jmath,\beta}(f(t_1, \ldots, t_n)) &= \jmath(f)(val_{\jmath,\beta}(t_1), \ldots, val_{\jmath,\beta}(t_n)) \end{aligned}$$

Definition 12 (Semantics of formulas)**.** *The semantics of a formula $f \in Form_S$ depending on an interpretation $\jmath$ and a valuation of variables β is given by a valuation function*

$$val_{\jmath,\beta}(F) : (Var \times D_{\mathcal{M}}) \to \mathbb{B}$$

This function is inductivly defined by

$$\begin{aligned}
val_{\jmath,\beta}(p(t_1,\ldots,t_n)) &= \jmath(p)(val_{\jmath,\beta}(t_1),\ldots,val_{\jmath,\beta}(t_n)) \\
val_{\jmath,\beta}(\neg F) = true &\quad \textit{iff} \quad val_{\jmath,\beta}(F) = false \\
val_{\jmath,\beta}(F \wedge G) = true &\quad \textit{iff} \quad val_{\jmath,\beta}(F) = true \textit{ and } val_{\jmath,\beta}(G) = true \\
val_{\jmath,\beta}(F \vee G) = true &\quad \textit{iff} \quad val_{\jmath,\beta}(F) = true \textit{ or } val_{\jmath,\beta}(G) = true \\
val_{\jmath,\beta}(F \to G) = true &\quad \textit{iff} \quad val_{\jmath,\beta}(F) = false \textit{ or } val_{\jmath,\beta}(G) = true \\
val_{\jmath,\beta}(F \leftrightarrow G) = true &\quad \textit{iff} \quad val_{\jmath,\beta}(F) = val_{\jmath,\beta}(G) \\
val_{\jmath,\beta}(\forall X F) = true &\quad \textit{iff} \quad \textit{for all values } d \in D_{\mathcal{M}}\ val_{\jmath,\beta[X\mapsto d]}(F) = true \textit{ holds} \\
val_{\jmath,\beta}(\exists X F) = true &\quad \textit{iff} \quad \textit{for one } d \in D_{\mathcal{M}}\ val_{\jmath,\beta[X\mapsto d]}(F) = true \textit{ holds}
\end{aligned}$$

1.5 KeY

The KeY project [ABB+05, BHS07, KeY07] was created from the idea of combining formal methods and software design. It is intended to integrate design and implementation of software systems with formal specification and formal verification "as seamlessly as possible" [BHS07]. For this purpose the KeY project was integrated with Borland Together [BSC06] as well as Eclipse [EFI05]. The user can enter his system specifications using the Object Constraint Language (OCL) [WK03] or the Java Modelling Language (JML) [LPC+07]. For verifying the specification, it is translated into a dynamic logic called JAVA CARD DL [Bec01]. With this logic one can express properties of a subset of JAVA programs (JAVA CARD [Che00]). The logic has been extended in order to provide an opportunity to verify JAVA programs that do not use dynamic class loading and floating point types [BHS07]. An interactive theorem prover is used in the verification process. This theorem prover component is a point of major strategic importance for us. Since we want to develop a theorem prover for $\mathsf{d}\mathcal{L}$ which is a dynamic logic as well, we decided that it might be easier to alter KeY instead of extending a first-order theorem prover, or develop a higher-order translation for $\mathsf{d}\mathcal{L}$ to prove properties with a higher-order theorem prover (like Isabelle [Pau94]). Another reason was that we expected better performance results, as KeY can already handle a dynamic logic.

The satisfaction problem of FOL is undecidable as the halting problem of a Turing machine with an empty band can be reduced to the satisfaction problem of FOL formulas [Coo71]. Therefore it is not possible to construct a fully automatic prover for FOL. This problem propagates to $\mathsf{d}\mathcal{L}$ as it is a conservative extension of FOL. There are algorithms to semi-decide the satisfaction problem, but an interactive approach can cover a bigger subset of $\mathsf{d}\mathcal{L}$ formulas.

Chapter 2

Differential Dynamic Logic

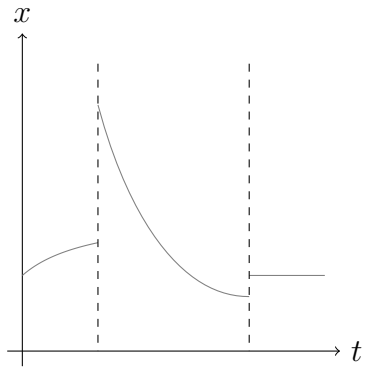
2.1 Overview

The differential dynamic logic ($\mathsf{d}\mathcal{L}$) is a dynamic logic for describing hybrid systems [Pla07c, Pla07b, Pla07a]. As a dynamic logic it has two relevant parts. On the one hand we have the classical first-order part. On the other hand we have the program world in the modalities. The programs in $\mathsf{d}\mathcal{L}$ are regular combinations of discrete state changes and continuous evolutions of the system variables.

Discrete state changes are possible using assignments like `x:=5`. This for example is used to model states of a controller component.

Continuous evolutions can be modelled using systems of differential equations and invariants. The continuous evolutions of variables are e.g. used to model the driving behavior of a train.

Example 1. *With the combination of these two system behaviors we can e.g. model the following system behavior:*



The discontinuous points of this trajectory are marked with dashed lines. At these points discrete state changes are performed.

In this chapter we will present the formal syntax and semantics of $\mathsf{d}\mathcal{L}$ as well as a sequent calculus, to provide a decision procedure on a syntactical level.

2.2 Syntax

The syntax of $\mathsf{d}\mathcal{L}$ [Pla07c] formulas is given by the set $\mathcal{F}(\mathcal{V}, \mathcal{S}, \Sigma)$ where $\mathcal{V}$ is a set of real-valued logical variables, $\mathcal{S}$ is a set of real-valued program variables and Σ is a relation between function symbols or predicate symbols and their arities.

The names of logical variables, program variables, function symbols and predicate symbols are pairwise disjoint. To define the content of $\mathcal{F}(\mathcal{V}, \mathcal{S}, \Sigma)$ we need two other sets: $\mathcal{T}(\mathcal{V}, \mathcal{S}, \Sigma)$ is the set of valid terms and $\mathcal{H}(\mathcal{V}, \mathcal{S}, \Sigma)$ the set of hybrid programs. The sets are simultaneously inductively defined in definitions 13, 14, 15 and 16.

To make formulas easier to read, we will use uppercase letters to identify logical variables and lowercase ones for program variables.

Definition 13 (Signature for $\mathsf{d}\mathcal{L}$)**.** *A signature for* $\mathsf{d}\mathcal{L}$ *formulas is given by* Σ*, which is a relation between function and predicate symbols and their arity. It contains at least the pairs* $(0, 0)$*,* $(1, 0)$*,* $(+, 2)$*,* $(-, 2)$*,* $(\times, 2)$*,* $(\div, 2)$ $(=, 2)$*,* $(\leq, 2)$*,* $(<, 2)$*,* $(\geq, 2)$ *and* $(>, 2)$ *with their usual interpretation.*

Definition 14 (Terms)**.** $\mathcal{T}(\mathcal{V}, \mathcal{S}, \Sigma)$ *is the set of all terms, which is the smallest set such that:*

- *if* $X \in \mathcal{V}$*, then* $X \in \mathcal{T}(\mathcal{V}, \mathcal{S}, \Sigma)$
- *if* $x \in \mathcal{S}$*, then* $x \in \mathcal{T}(\mathcal{V}, \mathcal{S}, \Sigma)$
- *if* $f \in \Sigma$ *then* $f(\theta_1, \ldots, \theta_n) \in \mathcal{T}(\mathcal{V}, \mathcal{S}, \Sigma)$ *where* f *is a function symbol with arity* n *and* $\theta_1, \ldots, \theta_n \in \mathcal{T}(\mathcal{V}, \mathcal{S}, \Sigma)$*.*

Definition 15 (Hybrid Program)**.** $\mathcal{H}(\mathcal{V}, \mathcal{S}, \Sigma)$ *is the set of all hybrid programs, which is the smallest set such that:*

- *if* $(x \in \mathcal{S}$ *and* $\theta \in \mathcal{T}(\mathcal{V}, \mathcal{S}, \Sigma))$*, then*$(x := \theta) \in \mathcal{H}(\mathcal{V}, \mathcal{S}, \Sigma)$
- *if* $(x \in \mathcal{S})$*, then* $(x := *) \in \mathcal{H}(\mathcal{V}, \mathcal{S}, \Sigma)$
- *if* $(x \in \mathcal{S}$*,* $\theta \in \mathcal{T}(\mathcal{V}, \mathcal{S}, \Sigma)$ *and* $\xi \in \mathcal{F}(\mathcal{V}, \mathcal{S}, \Sigma))$ *is a quantifier-free first-order formula, then* $(\dot{x} = \theta \wedge \xi) \in \mathcal{H}(\mathcal{V}, \mathcal{S}, \Sigma)$
- *if* $\varphi \in \mathcal{F}(\mathcal{V}, \mathcal{S}, \Sigma)$*, then* $(?\varphi) \in \mathcal{H}(\mathcal{V}, \mathcal{S}, \Sigma)$*, where* φ *is a quantifier-free first-order formula*
- *if* $\alpha, \gamma \in \mathcal{H}(\mathcal{V}, \mathcal{S}, \Sigma)$*, then* $(\alpha; \gamma) \in \mathcal{H}(\mathcal{V}, \mathcal{S}, \Sigma)$
- *if* $\alpha, \gamma \in \mathcal{H}(\mathcal{V}, \mathcal{S}, \Sigma)$*, then* $(\alpha \cup \gamma) \in \mathcal{H}(\mathcal{V}, \mathcal{S}, \Sigma)$
- *if* $\alpha \in \mathcal{H}(\mathcal{V}, \mathcal{S}, \Sigma)$*, then* $(\alpha^*) \in \mathcal{H}(\mathcal{V}, \mathcal{S}, \Sigma)$

Hybrid programs are regular combinations of $(x := \theta)$ and $(x := *)$ to express discrete state changes, the latter for a random one, i.e. after execution of $(x := *)$ the value of x is an arbitrary real number, $(\dot{x} = \theta \wedge \xi)$ for modelling continuous evolutions of a variable with invariant behavior ξ and state assertions that can be expressed using $(?\varphi)$. The operators for the regular combination are the sequential composition $(;)$, the non-deterministic choice $(\cup)$ and the non-deterministic repetition $(^*)$.

Definition 16 (Formulas)**.** *The set $\mathcal{F}(\mathcal{V},\mathcal{S},\Sigma)$ of formulas is the smallest set such that:*

- *if $((p,n)\in\Sigma$ and $\theta_i\in\mathcal{T}(\mathcal{V},\mathcal{S},\Sigma))$, then $p(\theta_1,\ldots,\theta_n)\in\mathcal{F}(\mathcal{V},\mathcal{S},\Sigma)$ where p is a predicate symbol.*
- *if $(\phi\in\mathcal{F}(\mathcal{V},\mathcal{S},\Sigma))$, then $(\neg\phi\in\mathcal{F}(\mathcal{V},\mathcal{S},\Sigma))$*
- *if $(\phi,\psi\in\mathcal{F}(\mathcal{V},\mathcal{S},\Sigma))$, then $(\phi\ op\ \psi)\in\mathcal{F}(\mathcal{V},\mathcal{S},\Sigma)$, where $op\in\{\wedge,\vee,\rightarrow,\leftrightarrow\}$*
- *if $(\phi\in\mathcal{F}(\mathcal{V},\mathcal{S},\Sigma)$ and $X\in\mathcal{V})$, then $(\forall X\phi),(\exists X\phi)\in\mathcal{F}(\mathcal{V},\mathcal{S},\Sigma)$*
- *if $(\phi\in\mathcal{F}(\mathcal{V},\mathcal{S},\Sigma)$ and $\alpha\in\mathcal{H}(\mathcal{V},\mathcal{S},\Sigma))$, then $([\alpha]\phi),(\langle\alpha\rangle\,\phi)\in\mathcal{F}(\mathcal{V},\mathcal{S},\Sigma)$*

2.3 Semantics

In this section we define the semantics of $\mathsf{d}\mathcal{L}$ [Pla07c]. For the valuation of terms and formulas we need an interpretation function for the function and predicate symbols, as well as a valuation of free variables, as well as a valuation of program variables . In definition 17 we define an interpretation function for the function and predicate symbols. Definition 18 provides a definition of a valuation function for free variables and a state defined in definition 19 is used for the valuation of program variables.

Definition 17 (Interpretations)**.** *An interpretation is a function $\jmath$ that assigns to each $(f,n)\in\Sigma$ where f is a function symbol a function*

$$\jmath(f):\mathbb{R}^n\rightarrow\mathbb{R}$$

and to each $(p,n)\in\Sigma$ if p is a predicate symbol a predicate

$$\jmath(f):\mathbb{R}^n\rightarrow\mathbb{B}$$

As we want to have standard mathematics as part of our logic we define our interpretations to contain the function symbols $\{+,-,\times,\div\}$ and the predicate symbols $\{<,\leq,=,\geq,>\}$ with the usual meanings.

Definition 18 (Valuation of free variables)**.** *A valuation of free variables is a function β that assigns to each variable $V\in\mathcal{V}$ a value $\beta(V)\in\mathbb{R}$.*

Definition 19 (State)**.** *A state is a function ν that assigns to each program variable $x\in\mathcal{S}$ a value $\nu(x)\in\mathbb{R}$. The set of all states is called $\Omega(\mathcal{S})$.*

Definition 20 (Modification of states)**.** *A modification $[x\mapsto r]$ of a state ν where x is a program variable and r is a real number results in a new state $\nu[x\mapsto r]$ that is identical to ν but $\nu[x\mapsto r](x)=r$ holds.*

Definition 21 (Rigidity)**.** *We define rigidity of terms as follows:*

- *A term is called* rigid *if it does not contain program variables.*
- *A term is called* non-rigid *if it does contain program variables.*

We define substitutions [Fit96] along the lines of the definition of substitutions in [Old02].

Definition 22 (Substitution)**.** *A* substitution *is a finite relation* $\Theta \subseteq \mathcal{V} \times \mathcal{T}(\mathcal{V}, \mathcal{S}, \Sigma)$*, written as* $\Theta = \{X_1/\theta_1, \ldots, X_n/\theta_n\}$*, with* $n \in \mathbb{N}_0$ *and*

- $X_1, \ldots, X_n$ *are pairwise disjoint*
- $\theta_1, \ldots, \theta_n$ *are rigid terms*
- *for all* $i \in \{1, \ldots, n\}$ $X_i \neq \theta_i$ *holds*

If there is only one variable to be substituted we abbreviate the substitution with $[X \mapsto \theta]$ *where* X *is the variable to substitute and* θ *is a term used as a substitution value.*

Definition 23 (Application of a substitution)**.** *Let* $t \in \mathcal{T}(\mathcal{V}, \mathcal{S}, \Sigma)$*,* $\alpha \in \mathcal{H}(\mathcal{V}, \mathcal{S}, \Sigma)$*,* $F \in \mathcal{F}(\mathcal{V}, \mathcal{S}, \Sigma)$ *and* $\Theta = \{X_1/\theta_1, \ldots, X_n/\theta_n\}$ *a substitution of the signature* Σ*.*

1. *The application of* Θ *to* t *results in a new term* $t\Theta$*, that is similar to* t *but all occurrences of the variables* $X_1, \ldots, X_n$ *are simultaneously replaced by the corresponding terms* $\theta_1, \ldots, \theta_n$*.*
2. *The application of* Θ *to* α *results in a new program* $\alpha\Theta$*, that is similar to* α *but all terms* t_α *occurring in* α *are replaced by the result of* $t_\alpha\Theta$*.*
3. *The application of* Θ *to* F *results in a new formula* $F\Theta$*, that is created by:*
   a) *renaming all occurrences of a variable* X_i *in a part of the formula with the form* $\exists XG$ *or* $\forall XG$ *where* G *is a formula with* $X \in \Theta_i$ *to a name that does not occur either in* F *nor in the substitution for* $i \in \{1, \ldots, n\}$
   b) *and replacing all remaining occurrences of the variables* $X_1, \ldots, X_n$ *by the corresponding terms* $\theta_1, \ldots, \theta_n$*.*

Definition 24 (Composition of substitutions)**.** *The* composition *of two substitutions*

$$\Theta_1 = \{X_1/s_1, \ldots, X_m/s_m\}$$
$$\Theta_2 = \{Y_1/t_1, \ldots, Y_n/t_n\}$$

is written as $\Theta_1\Theta_2$ *and can be computed by reducing the set*

$$\{X_1/s_1, \ldots, X_m/s_m, Y_1/t_1, \ldots, Y_n/t_n\}$$

by removing

1. *all pairs* Y_i/t_i *where* $Y_i \in \{X_1, \ldots, X_n\}$ *for* $i \in \{1, \ldots, n\}$

2. *all pairs* $X_i/s_i\Theta_2$ *where* $s_i\Theta_2 = X_i$ *for* $i \in \{1, \ldots, m\}$

Lemma 1. *The application of a substitution* $\Theta = \{X_1/\theta_1, \ldots, X_n/\theta_n\}$ *can be defined inductively over the syntax of terms, hybrid programs and formulas as:*

$$
\begin{aligned}
X\Theta &= \begin{cases} \theta_i & \text{if } X = X_i \text{ for one } i \in \{1, \ldots, n\} \\ X & \text{otherwise} \end{cases} \\
z\Theta &= z \\
(z := t)\Theta &= z := (t\Theta) \\
(z := *)\Theta &= z := * \\
(\dot{z} = t \wedge \xi)\Theta &= \dot{z} = (t\Theta) \wedge (\xi\Theta) \\
(\alpha;\ \gamma)\Theta &= \alpha\Theta;\ \gamma\Theta \\
(\alpha\ \cup\ \gamma)\Theta &= \alpha\Theta\ \cup\ \gamma\Theta \\
(\alpha^*)\Theta &= (\alpha\Theta)^* \\
(?\varphi)\Theta &= ?(\varphi\Theta) \\
f(t_1, \ldots, t_n)\Theta &= f(t_1\Theta, \ldots, t_n\Theta) \\
p(t_1, \ldots, t_n)\Theta &= p(t_1\Theta, \ldots, t_n\Theta) \\
(\neg\varphi)\Theta &= \neg(\varphi\Theta) \\
(\varphi\ op\ \psi)\Theta &= (\varphi\Theta\ op\ \psi\Theta) \\
&\quad \text{for } op \in \{\wedge, \vee, \rightarrow, \leftrightarrow\} \\
([\alpha]\varphi)\Theta &= [\alpha\Theta](\varphi\Theta) \\
(\langle\alpha\rangle\,\varphi)\Theta &= \langle\alpha\Theta\rangle\,(\varphi\Theta) \\
(\forall X\varphi)\Theta &= \forall y(\varphi[X \mapsto y]\Theta) \\
&\quad \text{where } y \notin var(\varphi) \cup dom(\Theta) \cup range(\Theta) \\
(\exists X\varphi)\Theta &= \exists y(\varphi[X \mapsto y]\Theta) \\
&\quad \text{where } y \notin var(\varphi) \cup dom(\Theta) \cup range(\Theta)
\end{aligned}
$$

We use the following symbols in this definition:

- X *and* Y *are logical variables*
- z *is a program variable*
- $t, t_1, \ldots, t_n$ *are terms*
- α, γ *are arbitrary hybrid programs*
- f *is a function symbol*
- p *is a predicate symbol*
- ξ, φ, ψ *are formulas*
- $var(\varphi)$ *donates all variables used in the formula* φ

- *dom*(Θ) *is the domain of* Θ*, i.e.* $\{X_1, \ldots, X_n\}$
- *range*(Θ) *is the range of* Θ*, i.e.* $\{\theta_1, \ldots, \theta_n\}$

Proof. The definition presented in lemma 1 produces the same results as the definition 23 beside the renaming of bound variables. As the constraint for the renaming is stronger in lemma 1 as in definition 23 and renaming of variables does not alter neither the satisfiability nor the validity of formulas, the definition in lemma 1 is sound. □

First we define a valuation function for terms as they are used inside formulas as well as inside hybrid programs.

Definition 25 (Valuation of terms)**.** *The valuation of terms with respect to a interpretation of rigid function symbols* $\jmath$*, valuation of free variables* β *and a state* ν *is given by the valuation function* $val_{\jmath,\beta}(\nu, \theta)$*:*

1. $val_{\jmath,\beta}(\nu, X) = \beta(X)$ *where* $X \in \mathcal{V}$
2. $val_{\jmath,\beta}(\nu, x) = \nu(x)$ *where* $x \in \mathcal{S}$
3. $val_{\jmath,\beta}(\nu, f(\theta_1, \ldots, \theta_n))) = \jmath(f)(val_{\jmath,\beta}(\nu, \theta_1), \ldots, val_{\jmath,\beta}(\nu, \theta_n))$

Second we define the semantics of hybrid programs. Hybrid programs are also referred to as system action. They are used to describe the control flow and behavior of a system.

Definition 26 (Semantics of hybrid programs)**.** *The semantics of hybrid programs is give by a valuation function* $\rho_{\jmath,\beta}$*. This is a transition relation that describes which states are reachable from the current state.*

1. $(\nu, \mu) \in \rho_{\jmath,\beta}(x := \theta)$ *iff* $\mu = \nu[x \mapsto val_{\jmath,\beta}(\nu, \theta)]$
2. $(\nu, \mu) \in \rho_{\jmath,\beta}(x := *)$ *iff* $\mu \in \{\nu[x \mapsto t] | t \in \mathbb{R}\}$
3. $(\nu, \mu) \in \rho_{\jmath,\beta}(\dot{x} = \theta \wedge \xi)$ *iff there is a function* $f : [0, r] \to \Omega(\mathcal{S})$ *with* $r \geq 0$ *such that* $\gamma_x(\zeta) = val_{\jmath,\beta}(f(\zeta), x)$ *is continuous on* $[0, r]$ *and differentiable of value* $\gamma_\theta(\zeta)$ *at each time* $\zeta \in]0, r[$*, while* γ_y *is constant for each* $y \neq x$ *and* $f(0) = \nu, f(r) = \mu$*. Also* $val_{\jmath,\beta}(f(\zeta), \xi) = true$ *holds for each* $\zeta \in]0, r[$*.*
4. $\rho_{\jmath,\beta}(?\varphi) = \{(\nu, \nu) \mid val_{\jmath,\beta}(\nu, \varphi) = true\}$
5. $\rho_{\jmath,\beta}(\alpha; \gamma) = \rho_{\jmath,\beta}(\alpha) \circ \rho_{\jmath,\beta}(\gamma) = \{(\nu, \mu) \mid (\nu, z) \in \rho_{\jmath,\beta}(\alpha), (z, \mu) \in \rho_{\jmath,\beta}(\gamma)$ *for some state* $z\}$*.*
6. $\rho_{\jmath,\beta}(\alpha \cup \gamma) = \rho_{\jmath,\beta}(\alpha) \cup \rho_{\jmath,\beta}(\gamma)$
7. $(\nu, \mu) \in \rho_{\jmath,\beta}(\alpha^*)$ *iff there are* $n \in \mathbb{N}$ *and* $\nu = \nu_0, \ldots, \nu_n = \mu$ *with* $(\nu_i, \nu_{i+1}) \in \rho_{\jmath,\beta}(\alpha)$ *for* $0 \leq i < n$

At last we define a function for the valuation of formulas.

Definition 27 (Valuation of formulas)**.** *The valuation function for formulas $val_{\jmath,\beta}(\nu,\varphi)$ with respect to an interpretation $\jmath$, a valuation of free variable β and a state ν is defined by:*

1. $val_{\jmath,\beta}(\nu, p(\theta_1,\ldots,\theta_n)) = \jmath(p)(val_{\jmath,\beta}(\nu,\theta_1),\ldots,val_{\jmath,\beta}(\nu,\theta_n))$
2. $val_{\jmath,\beta}(\nu,\neg\phi)$ = *true iff* $val_{\jmath,\beta}(\nu,\phi)$ = *false*
3. $val_{\jmath,\beta}(\nu,\phi\wedge\psi)$ = *true iff* $val_{\jmath,\beta}(\nu,\phi)$ = *true and* $val_{\jmath,\beta}(\nu,\psi)$ = *true*
4. $val_{\jmath,\beta}(\nu,\phi\vee\psi)$ = *true iff* $val_{\jmath,\beta}(\nu,\phi)$ = *true or* $val_{\jmath,\beta}(\nu,\psi)$ = *true*
5. $val_{\jmath,\beta}(\nu,\phi\rightarrow\psi)$ = *true iff* $val_{\jmath,\beta}(\nu,\phi)$ = *false or* $val_{\jmath,\beta}(\nu,\psi)$ = *true*
6. $val_{\jmath,\beta}(\nu,\phi\leftrightarrow\psi)$ = *true iff* $val_{\jmath,\beta}(\nu,\phi) = val_{\jmath,\beta}(\nu,\psi)$
7. $val_{\jmath,\beta}(\nu,\forall X\phi)$ = *true iff* $val_{\jmath,\beta[X\mapsto d]}(\nu,\phi)$ = *true for all* $d\in\mathbb{R}$
8. $val_{\jmath,\beta}(\nu,\exists X\phi)$ = *true iff* $val_{\jmath,\beta[X\mapsto d]}(\nu,\phi)$ = *true for some* $d\in\mathbb{R}$
9. $val_{\jmath,\beta}(\nu,[\alpha]\phi)$ = *true iff* $val_{\jmath,\beta}(\mu,\phi)$ = *true for all* μ *with* $(\nu,\mu)\in\rho_{\jmath,\beta}(\alpha)$
10. $val_{\jmath,\beta}(\nu,\langle\alpha\rangle\,\phi)$ = *true iff* $val_{\jmath,\beta}(\mu,\phi)$ = *true for some* μ *with* $(\nu,\mu)\in\rho_{\jmath,\beta}(\alpha)$

2.4 Sequent Calculus for d$\mathcal{L}$

In this section we present a sequent calculus for d$\mathcal{L}$ [Pla07c] that can be used to proof formulas valid by performing syntactic transformations.

2.4.1 Sequent Calculi

The first sequent calculus was developed in 1935 by Gerhard Gentzen and is a calculus for FOL [Gen35]. The central idea of a sequent calculus is to split the existing formulas into atomic ones and separate those atomic formulas which are *true* from those which are *false*. The calculus constructs a direct proof for the validity of a formula.

Definition 28 (Sequent)**.** $\Gamma\vdash\Delta$ *is a sequent with sets of formulas* Γ *and* Δ*.*

It is satisfiable *if for some interpretation $\jmath$, some valuation of free variable β and some state ν: $\jmath,\beta,\nu\not\models\varphi$ for some formula $\varphi\in\Gamma$ or $\jmath,\beta,\nu\models\psi$ for some formula $\psi\in\Delta$.*

The sequent is valid *if for all interpretations $\jmath$, all valuations of free variables β and all states ν $\jmath,\beta,\nu\not\models\varphi$ for some $\varphi\in\Gamma$ or $\jmath,\beta,\nu\models\psi$ for some formula $\psi\in\Delta$.*

Definition 29 (Calculus and Inference Rule)**.** *Assuming that $\mathcal{F}(\mathcal{V},\mathcal{S},\Sigma)$ is the set of formulas for a signature Σ, an* inference rule *is a relation $R\subseteq\mathcal{F}(\mathcal{V},\mathcal{S},\Sigma)^{n+1}$ with arity $(n+1)$ where $n\in\mathbb{N}_0$. The inference rule can either be written as*

$$R=\{(\varphi_1,\ldots,\varphi_n,\varphi_{n+1})|\ \textit{where}\ \psi(\varphi_1,\ldots,\varphi_n,\varphi_{n+1})\ \textit{holds}\}$$

or in the more common notation:

$$R: \frac{\varphi_1 \ \dots \ \varphi_n}{\varphi_{n+1}} \text{ where } \psi(\varphi_1, \dots, \varphi_n, \varphi_{n+1}) \text{ holds}$$

Where $\varphi_1, \dots, \varphi_n, \varphi_{n+1}$ *are schemata of formulas that have to satisfy* $\psi(\varphi_1, \dots, \varphi_n, \varphi_{n+1})$*. The schemata* $\varphi_1, \dots, \varphi_n$ *are called assumptions,* φ_{n+1} *is called conclusion.*

A finite set of inference rules is called calculus*.*

Definition 30 (Proof)**.** *A proof for the formula* φ *is a finite directed acyclic graph, such that*

- *there is only one root node*
- *each node is annotated with a sequent*
- *the annotated sequent of the root node is* $\vdash \varphi$
- *each node of the graph is annotated with an inference rule that relates the sequent of the node with the sequents of its decendants*
- *the number of descendants of a node is equal to the number of premisses of the annotated rule*

Even though the proof structure is in general a directed acyclic graph it is in most cases a tree.

The following example illustrates a common proof in a sequent calculus. The rules used in the proof are presented in section 2.4.2, but an intuition for the applied rules is already given at this place.

Example 2 (Proof Example)**.** *For verification of the propositional tautology* $\neg(A \vee B) \rightarrow (\neg A \wedge \neg B)$ *we construct a proof beginning with a node containing the sequent* $\vdash \neg(A \vee B) \rightarrow (\neg A \wedge \neg B)$*. The first operator we handle is the implication. As the sequent is defined as implication we can move the premiss to the left side of it. A negation is handled by moving the negated formula from one side of the sequent to another. The succeedents of a sequent are implicitly connected by a disjunction, so we can drop the disjoint operator. On the other hand a conjunction on the right side triggers branching of the proof. The two resulting goals can be closed after applying the negation rule again.*

$$\dfrac{\dfrac{\dfrac{*}{A \vdash A, B}\,R11}{\vdash \neg A, A, B}\,R2 \qquad \dfrac{\dfrac{*}{B \vdash A, B}\,R11}{\vdash \neg B, A, B}\,R2}{\vdash \neg A \wedge \neg B, A, B}\,R7$$

$$\dfrac{\vdash \neg A \wedge \neg B, A \vee B}{\vdash \neg A \wedge \neg B}\,R5$$

$$\dfrac{\neg(A \vee B) \vdash \neg A \wedge \neg B}{\vdash \neg(A \vee B) \rightarrow (\neg A \wedge \neg B)}\,R1$$

$$R3$$

2.4.2 Rules for Propositional Logic

The rules for handling the propositional part of the formulas are illustrated in figure 2.1. The rules are basically used for sorting the sequent, i.e. transforming the logic operators into the proof structure and eliminate negations.

The rules R1 and R2 eliminate negations. A negation in a sequent can be eliminated by moving the formula from one side to the other. To show that a sequent is satisfied, we have to show that there is a formula on the left side that is false or a formula on the right side that is true. If e.g. $\neg A$ occurs on the right side of the sequent, it is obvious that we can add A on the left side of the sequent, because if we can show that A is false, we have also shown that $\neg A$ is true.

The rules R6, R5 and R3 eliminate conjunctions, disjunctions and implication in the cases where no branch is necessary. A sequent is by definition an implication between the conjunction of the formulas on the left side and the disjunction of the formulas from the right side. As formulas on the left side are connected implicitly by a conjunction, we can remove conjunctions here. The same holds for the implicit disjunction and disjunctions on the right side. In total the sequent represents an implication. So if an implication occurs on the right side, we can move its premiss to the left side.

On the other hand the rules R7, R4 and R8 handle the cases where the proof branches. These cases are complementary to those described before. The branches of a proof are implicitly connected by a conjunction. If a conjunction occurs on the right side of the sequent, we have to show that both conjunctive elements are valid. This results in two new proof goals. For the complementary reason the proof branches if there is a disjunction on the left side. Here we have to show that in either the case where one disjunctive element is false or the other the proof can be closed. The case where both disjunctive elements are false is subsumed by these two cases. As an implication is by its semantics a special form of a disjunction the proof branches for an implication on the left side as well.

The equivalence is handled by the rules R9 and R10. It causes the proof to branch in both cases. For the case, that the equivalence occurs on the left side, it is necessary to show that the proof can be closed if the equivalence is valid, i.e. either both sides are true or both sides are false. If the equivalence would be false the sequent would trivially be valid. For the other case, we have to show that the formulas connected by the equivalence are complementary. This causes a branch as well, as again there are two cases to consider.

For closure of the proof the rule R11 is used. It can be applied if there is the same formula occurring on both sides of the sequent. A rule for closure is a rule without premisses, thus it leads to a leaf of the proof graph.

2.4.3 Rules for FOL

First-order logic can be handled by the propositional rules and additionally rules for handling quantifiers. These rules are shown in figure 2.3. As in $\mathsf{d}\mathcal{L}$ the quantifiers are real valued we use quantifier elimination [Tar51] to handle them. Tarski showed that for every FOL formula interpreted over the real numbers,

$$\text{(R1)}\quad \frac{\Gamma \vdash \Delta, A}{\Gamma, \neg A \vdash \Delta} \qquad \text{(R6)}\quad \frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}$$

$$\text{(R2)}\quad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta} \qquad \text{(R7)}\quad \frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}$$

$$\text{(R3)}\quad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \to B, \Delta} \qquad \text{(R8)}\quad \frac{\Gamma \vdash A, \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \to B \vdash \Delta}$$

$$\text{(R4)}\quad \frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta} \qquad \text{(R9)}\quad \frac{\Gamma, A, B \vdash \Delta \qquad \Gamma \vdash A, B, \Delta}{\Gamma, A \leftrightarrow B \vdash \Delta}$$

$$\text{(R5)}\quad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta} \qquad \text{(R10)}\quad \frac{\Gamma, A \vdash B, \Delta \qquad \Gamma, B \vdash A, \Delta}{\Gamma \vdash \Delta, A \leftrightarrow B}$$

$$\text{(R11)}\quad \frac{}{\Gamma, A \vdash A, \Delta}$$

- A and B are schemata of arbitrary $\mathsf{d}\mathcal{L}$ formulas
- Γ and Δ are arbitrary sets of schemata of $\mathsf{d}\mathcal{L}$ formulas

Figure 2.1: Propositional rules

there is an equivalent quantifier free formula.

We handle universal quantifiers on the right and existential quantifiers on the left side of the sequent using Skolemization [Fit96].

For a sound implementation of the Deskolemization rule (R14) that does not destroy completeness we need to assure that the quantifier elimination is applied to the inner-most quantifier, i.e. we must not eliminate a universal quantifier which quantifies a formula containing at least one existential quantifier and vice versa. This is necessary as the following formula is not universially valid:

$$(\forall X \exists Y \varphi(X, Y)) \to (\exists Y \forall X \varphi(X, Y))$$

Therefore we define a partial ordering on Skolem symbols to keep track of the quantifier order, which is retained in the parameters of the Skolem symbols.

Definition 31 (Partial ordering on Skolem symbols)**.** *We define a partial ordering on Skolem symbols as: sk_1 is smaller than sk_2 considering free variables ($sk_1 \prec_{fv} sk_2$) iff the set of parameters of sk_1 is a subset of the parameters of sk_2.*

In the following example we illustrate how this ordering can be used to assert that the right quantifier is eliminated.

Example 3. *If we got a formula like $\forall X \exists Y \forall Z \varphi(X, Y, Z)$ on the right side of the sequent, then the calculus works as follows (see also figure 2.2): The universal*

quantifier over X is dropped and a new function symbol sk_X is added with no parameters by rule R12. Afterwards the existential quantifier triggers a side deduction where Y is a free variable (R15). The rule R12 is applied again but the new symbol sk_Z gets the parameter Y as it is a free variable now. This is done to keep track of the quantifier order. At the point where rule R14 is applicable it would be possible to eliminate both, sk_X as well as sk_Z. For soundness it does not matter which one we eliminate as $\forall X \exists Y \forall Z \varphi(X,Y,Z) \rightarrow \forall X \forall Z \exists Y \varphi(X,Y,Z)$ holds and quantifiers of the same type are commutative. But as the reverse implication does not hold we may not be able to close the proof, when choosing the wrong symbol for elimination even if the formula is valid.

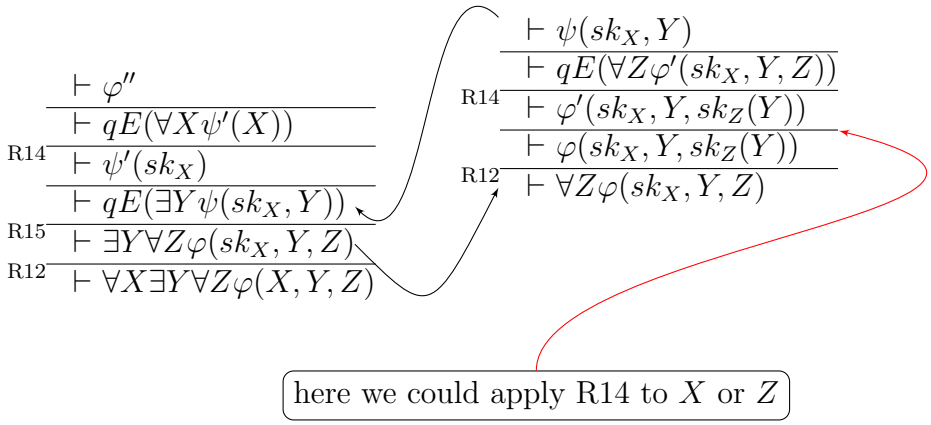


Figure 2.2: Example proof for the quantifier handling

The rule R14 works on a complete sequent as opposed to matching a fixed number of formulas inside the sequent. The rule separates the formulas of the sequent into four sets. Γ is the set of formulas on the left side of the sequent that do not contain the Skolem symbol sk_Y, whereas Γ' are those that do. The same holds for Δ and Δ' on the right side of the sequent.

It has to be said that using rules here that work on a single formula would also be correct but e.g. if we got a sequent like $\vdash \forall X(\varphi \vee \psi)$ it is easy to see that if we apply R12 the sequent $\vdash \varphi, \psi$ could be closed by applying R14 to φ and ψ separately but there does not have to be a solution even if the prior formula was valid.

As mentioned above, quantifier elimination can only be applied (by Mathematica) to first-order formulas. Thus we need to transform the quantified $\mathsf{d}\mathcal{L}$ formulas, possibly containing modalities specifying continuous evolutions of variables etc., into FOL. For the universal quantifier right case, we use Skolemization but for the existential quantifier right case we have two possibilities for this purpose. We could use a side deduction as suggested in figure 2.3 or we could try to integrate the transformation into the main proof as drafted in figure 2.5. This asymmetric solution has to be used, as the branches of a proof are implicitly connected by a

$$\text{(R12)}\quad \frac{\Gamma \vdash \varphi[X \mapsto sk_X(Y_1, \ldots, Y_n)], \Delta}{\Gamma \vdash \forall X \varphi, \Delta}$$

where sk_X new, Y_i free in φ

$$\text{(R13)}\quad \frac{\varphi[X \mapsto sk_X(Y_1, \ldots, Y_n)], \Gamma \vdash \Delta}{\exists X \varphi, \Gamma \vdash \Delta}$$

where sk_X new, Y_i free in φ

$$\text{(R14)}\quad \frac{\Gamma \vdash \Delta, qE(\forall X(\Gamma' \vdash \Delta'))}{\Gamma, \Gamma'[X \mapsto sk_Y(\ldots)] \vdash \Delta'[X \mapsto sk_Y(\ldots)], \Delta}$$

$$\text{(R15)}\quad \frac{qE(\exists X \bigwedge_i (\Gamma_i \vdash \Delta_i))}{\Gamma \vdash \Delta, \exists X \varphi}$$

$$\text{(R16)}\quad \frac{qE(\forall X \bigwedge_i (\Gamma_i \vdash \Delta_i))}{\Gamma, \forall X \varphi \vdash \Delta}$$

- Γ and Δ are arbitrary sets of schemata of $\mathsf{d}\mathcal{L}$ formulas
- φ is a schemata of a $\mathsf{d}\mathcal{L}$ formula
- sk_X and sk_Y are schemata of Skolem symbols
- X, Y_i for $i \in \{1, \ldots, n\}$ are schemata for variables
- The conditions for R14 are: sk_Y does not occur in $\Gamma \vdash \Delta$ and it is maximal considering $\prec_{fv}$. X is fresh, i.e. does not occur in $\Gamma' \vdash \Delta'$. All formulas in $\Gamma' \vdash \Delta'$ are quantifier free and first-order possibly prefixed with assignments.
- For the rules R15 and R16 the conditions are: X can be assumed to occur only in φ and a side deduction is started from $\Gamma \vdash \Delta, \varphi$ for rule R15 or $\Gamma, \varphi \vdash \Delta$ for rule R16 that yields to the first-order sequents $\Gamma_i \vdash \Delta_i$. (qE is the quantifier elimination).

Figure 2.3: First-order rules

conjunction. For the universal quantifier we can easily split this conjunction and handle it separately. For the existential quantifier this is not possible, as we need a simultaneous solution for all conjuncts.

Side deduction approach [Pla07c] The rules R16 and R15 trigger a side deduction (Figure 2.4) as a new proof obligation. The side deduction can be considered finished if all formulas on all leafs that are relevant, i.e. contain the quantified variable X, are first-order formulas. Afterwards the conjunction of all those leafs $(\Gamma_i \vdash \Delta_i)$ is returned to the proof where the side deduction was triggered from and the quantifier elimination is applied. The formulas that do not contain the variable X, can be abstracted as an atomic formula throughout the quantifier elimination as they have no influence on the choice of the variable. All steps performed in the side deduction have to be locally sound (see definition 37 on page 26). Otherwise the rules R16 and R15 could not be proven to be sound, as by applying a rule that is only globally sound within the side deduction we loose the information that the quantifier was an existential one.

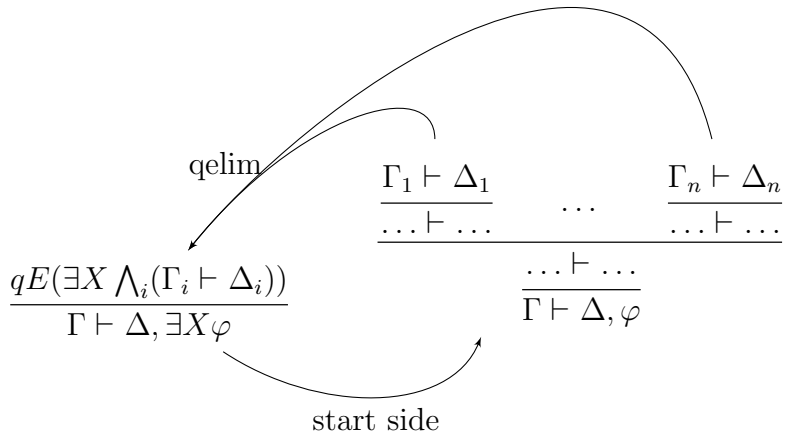


Figure 2.4: Side deduction

Simultaneous branch closure approach It seems very complicated to give sound rules that can reconstruct the existential quantifier (see figure 2.5), as it depends on the local soundness of all rules applied to the sequent since the removal of the existential quantifier. Formally these rules should result in two derivations. One that simply drops the quantifier resulting in a free variable and one to reintroduce it before quantifier elimination is applied. If we would use the rules drafted in figure 2.5, we would have the problem that the quantifier elimination has to consider all branches that contain the quantified variable. This

is necessary, as a closure of each branch alone does not yield a simultaneous solution for the implicit conjunction of the branches. For this, the rule would have to join all those branches. The proof would become a directed acyclic graph instead of a tree (see figure 2.6) which is uncommon for proofs in a sequent calculus. The condition for the reintroduction rule is that all rules applied in the deduction steps marked with ♠ are locally sound (see definition 37). The reason for the essential local soundness of these rules is the same as for the local soundness of the side deduction. The application of a rule that is not locally sound does not preserve the information that the quantifier was an existential one. We could always replace the existential quantifier by an universal one, as our domain is non-empty, but this would destroy completeness.

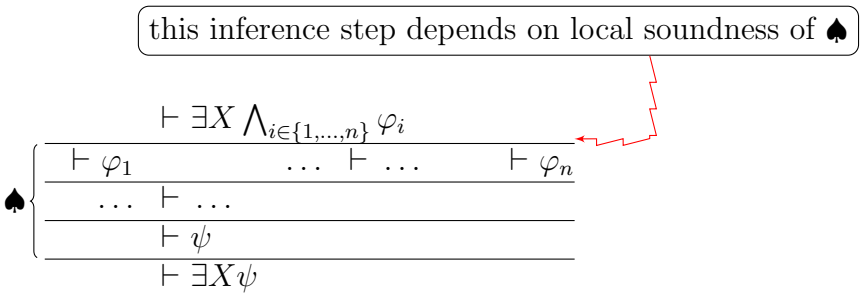


Figure 2.5: Alternative rule for existential quantifier on the right side

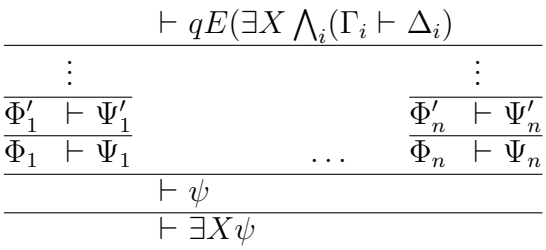


Figure 2.6: Proof structure with alternative rule for existential quantifier on the right side

2.4.4 Rules for Hybrid Programs

The rules for handling the modalities are presented in figure 2.7. These rules, except the rule R29, can be applied on either side of the sequent. We use this to reduce the number of rules, by writing these rules without a sequent sign.

We use $\langle\!|\alpha|\!\rangle$ to donate an abbreviation for either the modality $\langle\alpha\rangle$ or $[\alpha]$.

The basic principle is to split the modalities until we get elementary programs. The splitting is done by the rules R19, R20 and R21.

A sequential composition is split into two modalities by the rule R19. This is sound as the semantics of the sequential composition (definition 26) says that from the current state a state can be reached, by executing the first program, from which we can reach a state, by executing the second program, in which the postcondition holds.

The choice is handled differently depending on the modality type. If it is a box modality, we have to show that the postcondition holds if either the one or the other program is executed. For the diamond modality case it is sufficient to show that there is an execution of one of the programs that lead to a state where the postcondition is satisfied.

Loops cannot be eliminated so easily. They can be handled either using loop unrolling with rules R22 and R23 or by providing an invariant using the induction rule R29.

The invariant rule proves that a given invariant is valid in the current state, that it is preserved during the execution of the loop body and that it is strong enough to imply the postcondition. This leads to a sufficient description of the loop behavior. It may be noted that the current context is only used for the branch that shows that the invariant is initially valid. This branch is necessary as the loop could be executed zero times. To show that the invariant is preserved by the loop body we must not use the context formulas, as the state may already be changed by the loop before. Also for showing that the invariant can imply the postcondition, the state is only determined by the invariant not by contextual formulas.

The elementary programs are handled using the rules R17 and R18 for state assertions, R24 for assignments, R25 and R26 for random assignments as well as R27 and R28 for continuous evolutions.

State assertions are dragged to the logic level. The semantics of a state assertion says that it is a "no operation" operation if it is true and that there is no successor state if it is false. For the box modality case this leads to an implication with the state assertion as premiss and the postcondition as conclusion. This is expressed in rule R18. For the diamond modality case the situation is different. The semantics of the diamond modality forces that there is at least one execution of the program that leads to a state where the postcondition holds. This means that the state assertion as well as the postcondition have to be valid to satisfy the formula. This is expressed in the rule R17.

Assignments are applied by the rule R24 if they are applicable. The handling of assignments is similar to those of updates in KeY [BHS07]. As the parallelism of updates in KeY is only syntactic sugar, we decided to leave the assignments sequential.

Definition 32 (Application of assignments)**.** *An assignment $x := \theta$ is* applicable *to a formula φ if either*

- *the toplevel operator of φ is a first-order operator*
- *or φ has the form $\langle\!\langle \alpha \rangle\!\rangle \psi$ and α does not modify the value of x and no program variable is modified by α that occurs in θ.*

The application of the assignment $\mathcal{U}$ *is defined inductively. Applying an assignment to a term is inductively defined as:*

$$\begin{aligned}
\mathcal{U}(x,\theta,Y) &= Y \\
\mathcal{U}(x,\theta,z) &= \begin{cases} \theta & \text{if } z = x \\ z & \text{otherwise} \end{cases} \\
\mathcal{U}(x,\theta,f(t_1,\ldots,t_n)) &= f(\mathcal{U}(x,\theta,t_1),\ldots,\mathcal{U}(x,\theta,t_n))
\end{aligned}$$

Applying an assignment to a hybrid program is inductively defined as:

$$\begin{aligned}
\mathcal{U}(x,\theta,(z := t)) &= z := \mathcal{U}(x,\theta,t) \\
\mathcal{U}(x,\theta,(z := *)) &= z := * \\
\mathcal{U}(x,\theta,(\dot{z} = t \wedge \xi)) &= \dot{z} = \mathcal{U}(x,\theta,t) \wedge \mathcal{U}(x,\theta,\xi) \\
\mathcal{U}(x,\theta,?\varphi) &= ?(\mathcal{U}(x,\theta,\varphi)) \\
\mathcal{U}(x,\theta,(\alpha;\ \gamma)) &= \begin{cases} \mathcal{U}(x,\theta,\alpha);\ \mathcal{U}(x,\theta,\gamma) & \text{if } u(\alpha) \text{ and } u(\gamma) \\ x := \theta;\ \alpha;\ \gamma & \text{otherwise} \end{cases} \\
\mathcal{U}(x,\theta,(\alpha\ \cup\ \gamma)) &= \begin{cases} \mathcal{U}(x,\theta,\alpha)\ \cup\ \mathcal{U}(x,\theta,\gamma) & \text{if } u(\alpha) \text{ and } u(\gamma) \\ x := \theta; (\alpha\ \cup\ \gamma) & \text{otherwise} \end{cases} \\
\mathcal{U}(x,\theta,(\alpha^*)) &= \begin{cases} \mathcal{U}(x,\theta,\alpha)^* & \text{if } u(\alpha) \\ x := \theta; \alpha^* & \text{otherwise} \end{cases}
\end{aligned}$$

The application of an assignment to a formula is inductively defined as:

$$\begin{aligned}
\mathcal{U}(x,\theta,p(t_1,\ldots,t_n)) &= p(\mathcal{U}(x,\theta,t_1),\ldots,\mathcal{U}(x,\theta,t_n)) \\
\mathcal{U}(x,\theta,(\neg\varphi)) &= \neg(\mathcal{U}(x,\theta,\varphi)) \\
\mathcal{U}(x,\theta,(\varphi\ op\ \psi)) &= (\mathcal{U}(x,\theta,\varphi)\ op\ \mathcal{U}(x,\theta,\psi)) \\
&\quad \text{for } op \in \{\wedge,\vee,\rightarrow,\leftrightarrow\} \\
\mathcal{U}(x,\theta,([\alpha]\varphi)) &= \begin{cases} [\mathcal{U}(x,\theta,\alpha)](\mathcal{U}(x,\theta,\varphi)) & \text{if } u(\alpha) \\ [x := \theta][\alpha]\varphi & \text{otherwise} \end{cases} \\
\mathcal{U}(x,\theta,(\langle\alpha\rangle\,\varphi)) &= \begin{cases} \langle\mathcal{U}(x,\theta,\alpha)\rangle\,(\mathcal{U}(x,\theta,\varphi)) & \text{if } u(\alpha) \\ [x := \theta]\,\langle\alpha\rangle\,\varphi & \text{otherwise} \end{cases} \\
\mathcal{U}(x,\theta,(\forall Y\varphi)) &= \forall Y(\mathcal{U}(x,\theta,\varphi)) \\
\mathcal{U}(x,\theta,(\exists Y\varphi)) &= \exists Y(\mathcal{U}(x,\theta,\varphi))
\end{aligned}$$

We use the following abbreviations

- $u(\alpha)$ *donates whether the assignment is applicable to the program* α *or not*
- Y *is a logical variable*

- *x and z are program variables*
- *f is a function symbol*
- *p is a predicate symbol*
- *$\theta, t, t_1, \ldots, t_n$ are terms*
- *α, γ are arbitrary hybrid programs*
- *ξ, φ, ψ are formulas*

It may be noted that we always use the box modality in this definition to represent the assignment if its not applicable to a subformula. This is done for convenience reasons. The semantics of the box and the diamond modality assignment is equivalent so it does not matter which modality type we use to reintroduce it.

On the other hand random assignments need different rules depending on the modality type. As the semantics of the box modality demand that every successor state of the modality satisfies the postcondition, a universal quantifier is introduced to describe the possible values for the randomly assigned variable. In case of a diamond modality the semantics demands that there is one execution that leads to a state satisfying the postcondition. Therefore the possible values of the variable are quantified with an existential quantifier.

The last elementary program is the continuous evolution. The differential equation is translated into a function that serves as solution of the system. This function depends on a newly introduced variable t that measures the execution time. In case of a box modality, for the same reason as above, the postcondition has to hold for every possible execution time. In case of diamond modality it is sufficient if there is a satisfying execution time. The invariant expression has, in both cases, to be satisfied for every possible execution below the chosen one. This is expressed in the rules R27 for the diamond case and R28 for the box case.

2.4.5 Soundness Proofs

In this section we define soundness for a calculus and different types of soundness for inference rules. Also we will show that the altered version of the calculus is sound.

To define soundness of a calculus, we need different types of relations. These relations are used to express validity of formulas. First we define a global satisfaction relation.

Definition 33 (Satisfaction Relation)**.** *An interpretation $\jmath$, a valuation of free variables β and a state ν* satisfy *a formula φ written as $\jmath, \beta, \nu \models \varphi$ iff $val_{\jmath,\beta}(\nu, \varphi) = true$.*

With this definition we can define the terms satisfiability and validity.

$$\text{(R17)}\quad \frac{\varphi \wedge \psi}{\langle ?\varphi \rangle\, \psi} \qquad \text{(R19)}\quad \frac{\langle\!\langle \alpha \rangle\!\rangle\, \langle\!\langle \gamma \rangle\!\rangle\, \varphi}{\langle\!\langle \alpha; \gamma \rangle\!\rangle\, \varphi} \qquad \text{(R21)}\quad \frac{[\alpha]\, \varphi \wedge [\gamma]\, \varphi}{[\alpha \cup \gamma]\, \varphi}$$

$$\text{(R18)}\quad \frac{\varphi \rightarrow \psi}{[?\varphi]\, \psi} \qquad \text{(R20)}\quad \frac{\langle \alpha \rangle\, \varphi \vee \langle \gamma \rangle\, \varphi}{\langle \alpha \cup \gamma \rangle\, \varphi} \qquad \text{(R22)}\quad \frac{\varphi \vee \langle \alpha; \alpha^* \rangle\, \varphi}{\langle \alpha^* \rangle\, \varphi}$$

$$\text{(R23)}\quad \frac{\varphi \wedge [\alpha; \alpha^*]\, \varphi}{[\alpha^*]\, \varphi} \qquad \text{(R25)}\quad \frac{\forall T [x := T] \varphi}{[x := *]\, \varphi}$$

$$\text{(R24)}\quad \frac{\mathcal{U}(x, \theta, \varphi)}{\langle\!\langle x := \theta \rangle\!\rangle\, \varphi} \qquad \text{(R26)}\quad \frac{\exists T [x := T] \varphi}{\langle x := * \rangle\, \varphi}$$

$$\text{(R27)}\quad \frac{\exists\, T \geq 0\, \forall\, 0 < \tilde{T} < T \left\langle x := y_v(\tilde{T}) \right\rangle \xi \rightarrow \langle x := y_v(T) \rangle\, \varphi}{\langle \dot{x} = \theta \wedge \xi \rangle\, \varphi}$$

$$\text{(R28)}\quad \frac{\forall\, T \geq 0\, \forall\, 0 < \tilde{T} < T \left[x := y_v(\tilde{T}) \right] \xi \rightarrow [x := y_v(T)]\, \varphi}{[\dot{x} = \theta \wedge \xi]\, \varphi}$$

$$\text{(R29)}\quad \frac{\Gamma \vdash \mathcal{A} p, \Delta \qquad p \vdash [\alpha]\, p \qquad p \vdash \varphi}{\Gamma \vdash \mathcal{A}\, [\alpha^*]\, \varphi, \Delta}$$

- Γ and Δ are arbitrary sets of schemata of $\mathsf{d}\mathcal{L}$ formulas
- φ, ψ, ξ and p are schemata of $\mathsf{d}\mathcal{L}$ formulas
- α and γ are schemata of hybrid programs
- θ is a schema of a term
- x is a schema for a program variable
- T and $\tilde{T}$ are schemata of a logical variables
- y_v is a schema for a function symbol
- $\mathcal{A}$ is a schema for a finite number of assignments within either box or diamond modalities

The side conditions are:

- the assignment in rule R24 must be applicable
- T is a fresh variable
- y_v is the solution of the initial value problem $(\dot{x} = \theta, x(0) = v)$

Figure 2.7: Rules for modalities

Definition 34 (Satisfiability and validity)**.** *A formula φ is* satisfiable *if for some interpretation $\jmath$, some valuation of free variables β and some state ν $\jmath, \beta, \nu \models \varphi$ holds.*

It is valid *if for all interpretation $\jmath$, all valuation of free variables β and all state ν $\jmath, \beta, \nu \models \varphi$ holds.*

Definition 35 (Consequence Relation)**.** *We say ψ is a (global)* consequence *of φ (written as $\varphi \models \psi$) iff for interpretations $\jmath$, for all valuations of free variables β and for all states ν $(\forall\beta\ \forall\nu\ \forall\jmath\ \ (\jmath, \beta, \nu \models \varphi)) \Rightarrow (\forall\beta\ \forall\nu\ \forall\jmath\ \ (\jmath, \beta, \nu \models \psi))$ holds.*

As we need a local consequence for some soundness proofs we define a special consequence relation for it.

Definition 36 (Not fully local consequence)**.** *We say $\varphi \models_{\beta,\nu} \psi$, i.e. ψ is a not fully local consequence of φ with respect to β and ν, iff for interpretations $\jmath$, valuations of free variables β and states ν $\forall\beta\ \forall\nu\ (\forall\jmath\ \ (\jmath, \beta, \nu \models \varphi) \Rightarrow \forall\jmath\ \ (\jmath, \beta, \nu \models \psi))$ holds.*

In [Pla07c] the author defined local consequence as $\varphi \models_\ell \psi$ iff $\forall\jmath\ \forall\beta\ \forall\nu\ (\jmath, \beta, \nu \models \varphi \Rightarrow \jmath, \beta, \nu \models \psi)$. We will show that this relation is a subset of our not fully local consequence relation, thus if a rule is sound with respect to $\models_\ell$ it is also sound with respect to $\models_{\beta,\nu}$.

Lemma 2 (Consequence relation subsets)**.** *The consequence relation $\models_\ell$ is a subset of $\models_{\beta,\nu}$ (written as $\models_\ell \subseteq \models_{\beta,\nu}$), i.e. $\varphi \models_\ell \psi$ implies $\varphi \models_{\beta,\nu} \psi$.*

Proof. We want to proof that

$$\forall\beta\ \forall\nu\ \forall\jmath\ ((\jmath, \beta, \nu \models \varphi) \Rightarrow (\jmath, \beta, \nu \models \psi)) \Rightarrow \forall\beta\ \forall\nu\ (\forall\jmath\ \ (\jmath, \beta, \nu \models \varphi) \Rightarrow \forall\jmath\ \ (\jmath, \beta, \nu \models \psi))$$

In prenix normal form this is:

$$\exists\beta\ \exists\nu\ \exists\jmath\ \forall\beta_2\ \forall\nu_2\ \exists\jmath_2\ \forall\jmath_3\ (((\jmath, \beta, \nu \models \varphi) \Rightarrow (\jmath, \beta, \nu \models \psi)) \Rightarrow ((\jmath_2, \beta_2, \nu_2 \models \varphi) \Rightarrow (\jmath_3, \beta_2, \nu_2 \models \psi)))$$

We can now transform the implications into disjunctions and get:

$$\exists\beta\ \exists\nu\ \exists\jmath\ \forall\beta_2\ \forall\nu_2\ \exists\jmath_2\ \forall\jmath_3\ (\jmath, \beta, \nu \models \varphi \vee \jmath, \beta, \nu \not\models \psi \vee \jmath_2, \beta_2, \nu_2 \not\models \varphi \vee \jmath_3, \beta_2, \nu_2 \models \psi)$$

If we assume that

$$\forall\beta_2\ \forall\nu_2\ \exists\jmath_2\ (\jmath_2, \beta_2, \nu_2 \not\models \varphi)$$

does not hold, the following proposition holds:

$$\exists\beta\ \exists\nu\ \forall\jmath\ (\jmath, \beta, \nu \models \varphi)$$

This means that

$$\exists\beta\ \exists\nu\ \exists\jmath\ (\jmath, \beta, \nu \models \varphi)$$

holds as well as the set of interpretations is non-empty.

□

With the previous definitions it is possible to define soundness of inference rules.

Definition 37 (Soundness of inference rules)**.** *We define* soundness of inference rules *as follows:*

1. *An inference rule*

$$R: \frac{\varphi_1 \ \dots \ \varphi_n}{\varphi_{n+1}} \text{ where } \psi(\varphi_1, \dots, \varphi_n, \varphi_{n+1}) \text{ holds}$$

is sound *if*

$$\Big(\bigwedge_{i\in\{1,\dots,n\}} \varphi_i\Big) \models \varphi_{n+1}$$

where $\psi(\varphi_1, \dots, \varphi_n, \varphi_{n+1})$ is satisfied holds.

2. *It is called* locally sound *if also*

$$\Big(\bigwedge_{i\in\{1,\dots,n\}} \varphi_i\Big) \models_{\beta,\nu} \varphi_{n+1}$$

holds.

3. *A calculus is called* sound *if all its inference rules are sound.*

Lemma 3 (Context of rules)**.** *The context $\Gamma \vdash \Delta$ can be ignored for proving the soundness of rules that do not change the context [Pla04].*

Proof. For proving soundness of rules, e.g. the soundness of a rule

$$\frac{\Gamma, \varphi_1 \vdash \Delta, \varphi_1' \ \dots \ \Gamma, \varphi_n \vdash \Delta, \varphi_n'}{\Gamma, \psi \vdash \Delta, \psi'}$$

we have to show that

$$\bigwedge_{i\in\{1,\dots,n\}} \Gamma, \varphi_i \vdash \Delta, \varphi_i' \models \Gamma, \psi \vdash \Delta, \psi'$$

Now let us consider the context. If Γ implies Δ the conjunction on the left side of this relation becomes true, as well as the sequent on the right side. So we can assume for the soundness proofs that the context does not yield a solution. Therefore it is sufficient to show

$$\bigwedge_{i\in\{1,\dots,n\}} \varphi_i \vdash \varphi_i' \models \psi \vdash \psi'$$

□

As we delay the applications of assignments, we have to show that the rules that are locally sound can also be applied if there is a prefix of assignments. We perform this proof along the lines of the proof for contextual lifting in [Pla04].

Lemma 4 (Assignment introduction)**.** *If the inference rule*

$$\frac{\Gamma_1 \vdash \Delta_1 \ \ldots \ \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}$$

is sound with respect to $\models_{\beta,\nu}$*, the inference rule*

$$\frac{[\![\mathcal{A}]\!]\,\Gamma_1 \vdash [\![\mathcal{A}]\!]\,\Delta_1 \ \ldots \ [\![\mathcal{A}]\!]\,\Gamma_n \vdash [\![\mathcal{A}]\!]\,\Delta_n}{[\![\mathcal{A}]\!]\,\Gamma \vdash [\![\mathcal{A}]\!]\,\Delta}$$

for some assignment $\mathcal{A}$ *is sound with respect to* $\models_{\beta,\nu}$ *as well.*

The sequent context Γ and Δ could also be written as $\varphi_1, \ldots, \varphi_n$ and $\psi_1, \ldots, \psi_m$. We use $[\![\mathcal{A}]\!]\,\Gamma$ and $[\![\mathcal{A}]\!]\,\Delta$ to donate abbreviations $[\![\mathcal{A}]\!]\,\varphi_1, \ldots, [\![\mathcal{A}]\!]\,\varphi_n$ and $[\![\mathcal{A}]\!]\,\psi_1, \ldots, [\![\mathcal{A}]\!]\,\psi_n$.

Proof. For convenience reasons we will show that if the rule

$$\frac{\Gamma' \vdash \Delta'}{\Gamma \vdash \Delta}$$

is locally sound so is the rule

$$\frac{[\![\mathcal{A}]\!]\,\Gamma' \vdash [\![\mathcal{A}]\!]\,\Delta'}{[\![\mathcal{A}]\!]\,\Gamma \vdash [\![\mathcal{A}]\!]\,\Delta}$$

This is sufficient to show for proving the soundness of lemma 4 as the possible branch could be expressed as conjunction over the elements Γ_i and disjunction over Δ_i for $i \in \{1, \ldots, n\}$.

1. First, we handle the case where $\Gamma = \Gamma' = \emptyset$. From the premiss we can conclude that

   $$\forall\beta \ \forall\nu \ (\forall\jmath \ \ (\jmath, \beta, \nu \models \ \vdash \Delta') \Rightarrow \forall\jmath \ \ (\jmath, \beta, \nu \models \ \vdash \Delta))$$

   holds. As the introduction of the assignment only alters the state we can conclude that for every state μ with $\mu\rho_{\jmath,\beta}(\mathcal{A})\nu$ the sequent $\vdash \langle\mathcal{A}\rangle\,\Delta$ is satisfied. As $\mathcal{A}$ is an assignment we can conclude that $\vdash [\mathcal{A}]\Delta$ is satisfied as well. This means that

   $$\forall\beta \ \forall\nu \ (\forall\jmath \ \ (\jmath, \beta, \nu \models \ \vdash [\![\mathcal{A}]\!]\,\Delta') \Rightarrow \forall\jmath \ \ (\jmath, \beta, \nu \models \ \vdash [\![\mathcal{A}]\!]\,\Delta))$$

   holds as well.
2. In case of Γ and Γ' are arbitrary we can use the definition of the sequent (definition 28) to conclude that if the premiss holds the rule

   $$\frac{\vdash \neg\Gamma', \Delta'}{\vdash \neg\Gamma, \Delta}$$

   is locally sound as well. Now this case equivalent to the first case.

□

Lemma 5 (Soundness of the calculus)**.** *All rules presented in figures 2.1, 2.3 and 2.7 are sound. All rules presented beside the rule R29 are locally sound as well.*

Proof. With lemma 2 (page 25) we can conclude that all rules that are proven to be locally sound in [Pla07c] are also sound with respect to our not fully local consequence relation 36. □

We will present proofs for the rules that were changed compared to the calculus in [Pla07c].

Soundness of rule R9. The rule R9 is defined as:

$$\text{(R9)} \quad \frac{\Gamma, A, B \vdash \Delta \qquad \Gamma \vdash A, B, \Delta}{\Gamma, A \leftrightarrow B \vdash \Delta}$$

We want to proof that this rule is locally sound. This means:

$$\Gamma, A, B \vdash \Delta \wedge \Gamma \vdash A, B, \Delta \models_{\beta,\nu} \Gamma, A \leftrightarrow B \vdash \Delta$$

Using the definition of the sequent (definition 28) and the definition of the local consequence relation (definition 36) and the context lemma (lemma 3) we get:

$$\forall\beta\ \forall\nu\ (\forall\jmath\ \ (\jmath, \beta, \nu \models \neg(A \wedge B) \wedge (A \vee B)) \Rightarrow \forall\jmath\ \ (\jmath, \beta, \nu \models \neg(A \leftrightarrow B)))$$

This can be transformed to

$$\forall\beta\ \forall\nu\ (\forall\jmath\ \ (\jmath, \beta, \nu \models (\neg A \vee \neg B) \wedge (A \vee B)) \Rightarrow \forall\jmath\ \ (\jmath, \beta, \nu \models \neg(A \leftrightarrow B)))$$

using one of de Morgan's laws [Fit96]

$$\neg(A \wedge B) \equiv (\neg A \vee \neg B)$$

The semantic definition of $A \leftrightarrow B$ says that either A and B are $true$, or both are $false$ (definition 27). This means that

$$A \leftrightarrow B \equiv ((A \wedge B) \vee (\neg A \wedge \neg B))$$

holds. Using this equivalence we get

$$\forall\beta\ \forall\nu\ (\forall\jmath\ \ (\jmath, \beta, \nu \models (\neg A \vee \neg B) \wedge (A \vee B)) \Rightarrow \forall\jmath\ \ (\jmath, \beta, \nu \models \neg((A \wedge B) \vee (\neg A \wedge \neg B))))$$

Now we apply de Morgan's laws again:

$$\neg((A \wedge B) \vee (\neg A \wedge \neg B)) \equiv (\neg(A \wedge B) \wedge \neg(\neg A \wedge \neg B)) \equiv (\neg A \vee \neg B) \wedge (A \vee B)$$

$$\forall\beta\ \forall\nu\ (\forall\jmath\ \ (\jmath, \beta, \nu \models (\neg A \vee \neg B) \wedge (A \vee B)) \Rightarrow \forall\jmath\ \ (\jmath, \beta, \nu \models (\neg A \vee \neg B) \wedge (A \vee B)))$$

This is valid as both sides of the implication are the same.

□

Soundness of rule R10. The rule R10 is defined as:

$$\text{(R10)} \quad \frac{\Gamma, A \vdash B, \Delta \qquad \Gamma, B \vdash A, \Delta}{\Gamma \vdash \Delta, A \leftrightarrow B}$$

The local soundness of this rule can be performed along the lines of the soundness proof for rule R9. We have to show that

$$\Gamma, A \vdash B, \Delta \land \Gamma, B \vdash A, \Delta \models_{\beta,\nu} \Gamma \vdash \Delta, A \leftrightarrow B$$

holds. Using the definition of the sequent (definition 28) and the definition of the local consequence relation (definition 36) and the context lemma (lemma 3) we get:

$$\forall\beta \ \forall\nu \ (\forall\jmath \ \ (\jmath, \beta, \nu \models (A \to B) \land (B \to A)) \Rightarrow \forall\jmath \ \ (\jmath, \beta, \nu \models A \leftrightarrow B))$$

The implication is defined as either the premisses are false or the conclusion is true (definition 27). This means that the following equivalence holds

$$(\varphi \to \psi) \equiv (\neg\varphi \lor \psi)$$

Using this equivalence we can conclude

$$\forall\beta \ \forall\nu \ (\forall\jmath \ \ (\jmath, \beta, \nu \models (\neg A \lor B) \land (\neg B \lor A)) \Rightarrow \forall\jmath \ \ (\jmath, \beta, \nu \models A \leftrightarrow B))$$

There are exactly two valuations for A and B that satisfy $((\neg A \lor B) \land (\neg B \lor A))$. Either A and B are both *true*, or they are both $false$. Using the definition of the equivalence $\leftrightarrow$ (definition 27) we can unify both sides of the implication. □

Lemma 6 (Soundness of rule R24)**.** *The rule R24 is sound.*

The rule R24 is defined as:

$$\text{(R24)} \quad \frac{\mathcal{U}(x, \theta, \varphi)}{\langle\!\langle x := \theta \rangle\!\rangle \varphi}$$

We want to show that this rule is locally sound. Therefore we need to proof that for all valuations of free variables β and all states ν if the premisses of the rule are valid under all interpretations $\jmath$ so are its conclusions. We have to show that

$$\mathcal{U}(x, \theta, \varphi) \models_{\beta,\nu} [x := \theta] \varphi$$

holds. Using definition 36 we get:

$$\forall\beta \ \forall\nu \ (\forall\jmath \ \ (\jmath, \beta, \nu \models \mathcal{U}(x, \theta, \varphi)) \Rightarrow \forall\jmath \ \ (\jmath, \beta, \nu \models [x := \theta] \varphi))$$

As it is easier to prove we show that the even stronger relation holds:

$$\forall\jmath \ \forall\beta \ \forall\nu \ (\jmath, \beta, \nu \models \mathcal{U}(x, \theta, \varphi) \Rightarrow \jmath, \beta, \nu \models [x := \theta] \varphi)$$

We prove this using induction over the structure of φ.
In this proof we use the following abbreviations

- $u(\alpha)$ donates whether the assignment is applicable to the program α or not
- Y is a logical variable
- x and z are program variables
- f is a function symbol
- p is a predicate symbol
- $\theta, t, t_1, \ldots, t_n$ are terms
- α, γ are arbitrary hybrid programs
- ξ, φ, ψ are formulas

The semantic of the assignment $[x := \theta]$ is defined as modification of the current state by assigning the value of θ to the program variable x. Therefore we have to show that the update application assigns the same value to x in every case. We name the state that is reached by executing the assignment μ.

We have to show three propositions:

1. The valuation of terms is identical

$$val_{\jmath,\beta}(\nu, \mathcal{U}(x, \theta, t)) = val_{\jmath,\beta}(\mu, t) \tag{2.1}$$

2. The set of successor states for the hybrid programs is the same

$$\rho_{\jmath,\beta}(\mathcal{U}(x, \theta, \alpha))(\nu) = \rho_{\jmath,\beta}(\alpha)(\mu) \tag{2.2}$$

3. The valuation of formulas is identical

$$val_{\jmath,\beta}(\nu, \mathcal{U}(x, \theta, \varphi)) = val_{\jmath,\beta}(\mu, \varphi) \tag{2.3}$$

Proof. From proposition 3 we can conclude that

$$\forall \jmath \ \forall \beta \ \forall \nu \ \ (\jmath, \beta, \nu \models \mathcal{U}(x, \theta, \varphi) \ \Rightarrow \ \jmath, \beta, \nu \models [x := \theta]\, \varphi)$$

holds. □

First we show that proposition 1 holds.

Proof. From the definition of the assignment application (definition 32) we can derive the following cases.

For terms we get the following valuations in ν.

$$\begin{aligned} val_{\jmath,\beta}(\nu, \mathcal{U}(x, \theta, Y)) &= val_{\jmath,\beta}(\nu, Y) \\ val_{\jmath,\beta}(\nu, \mathcal{U}(x, \theta, z)) &= \begin{cases} val_{\jmath,\beta}(\nu, \theta) & \text{if } z = x \\ val_{\jmath,\beta}(\nu, z) & \text{otherwise} \end{cases} \end{aligned}$$

For this three cases we can easily see that the terms in the state get the same value in μ. In the case of a logical variable the value is not determined by the state. The states ν and μ are identical for all program variables beside x such that if $z \neq x$, the value is same. For the case that the term is x its value is θ in both cases.

$$\begin{aligned}
& val_{\mathcal{I},\beta}(\nu, \mathcal{U}(x,\theta, f(t_1,\ldots,t_n))) \\
= \;& val_{\mathcal{I},\beta}(\nu, f(\mathcal{U}(x,\theta,t_1),\ldots,\mathcal{U}(x,\theta,t_n))) \\
= \;& \mathcal{I}(f)(val_{\mathcal{I},\beta}(\nu,\mathcal{U}(x,\theta,t_1)),\ldots,val_{\mathcal{I},\beta}(\nu,\mathcal{U}(x,\theta,t_n))) \\
\overset{(2.1)}{=} \;& \mathcal{I}(f)(val_{\mathcal{I},\beta}(\mu,t_1),\ldots,val_{\mathcal{I},\beta}(\mu,t_n)) \\
= \;& val_{\mathcal{I},\beta}(\mu, f(t_1,\ldots,t_n))
\end{aligned}$$

□

Next we show that proposition 2 holds.

Proof. Using proposition 1 and the definition of the application of assignments (definition 32) we can easily show that the states that are reachable from ν after applying the assignment are the same as the states that are reachable from the state μ. For the cases of differential equations with invariant and state assertions, we also need proposition 3 to show this.

$$\begin{aligned}
\rho_{\mathcal{I},\beta}(\mathcal{U}(x,\theta,(z:=t)))(\nu) &= \rho_{\mathcal{I},\beta}(z:=\mathcal{U}(x,\theta,t))(\nu) \overset{(2.1)}{=} \rho_{\mathcal{I},\beta}(z:=t)(\mu) \\
&= \rho_{\mathcal{I},\beta}(x:=\theta; z:=t)(\nu) \\
\rho_{\mathcal{I},\beta}(\mathcal{U}(x,\theta,(z:=*)))(\nu) &= \rho_{\mathcal{I},\beta}(z:=*)(\nu) = \rho_{\mathcal{I},\beta}(z:=*)(\mu) \\
&= \rho_{\mathcal{I},\beta}(x:=\theta; z:=*)(\nu) \\
\rho_{\mathcal{I},\beta}(\mathcal{U}(x,\theta,(\dot{z}=t\wedge\xi)))(\nu) &= \rho_{\mathcal{I},\beta}(\dot{z}=\mathcal{U}(x,\theta,t)\wedge\mathcal{U}(x,\theta,\xi))(\nu) \\
&\overset{(2.1)\wedge(2.3)}{=} \rho_{\mathcal{I},\beta}(\dot{z}=t\wedge\xi)(\mu) \\
&= \rho_{\mathcal{I},\beta}(x:=\theta;\dot{z}=t\wedge\xi)(\nu) \\
\rho_{\mathcal{I},\beta}(\mathcal{U}(x,\theta,?\varphi))(\nu) &= \rho_{\mathcal{I},\beta}(?(\mathcal{U}(x,\theta,\varphi)))(\nu) \overset{(2.3)}{=} \rho_{\mathcal{I},\beta}(?\varphi)(\mu) \\
&= \rho_{\mathcal{I},\beta}(x:=\theta;?\varphi)(\nu)
\end{aligned}$$

Now lets consider compound programs.

$$\rho_{\mathcal{I},\beta}(\mathcal{U}(x,\theta,(\alpha;\ \gamma)))(\nu) = \begin{cases} \rho_{\mathcal{I},\beta}(\mathcal{U}(x,\theta,\alpha);\ \mathcal{U}(x,\theta,\gamma))(\nu) & \text{if } u(\alpha) \text{ and } u(\gamma) \\ \rho_{\mathcal{I},\beta}(x:=\theta;\ \alpha;\ \gamma)(\nu) & \textit{otherwise} \end{cases}$$

$$\rho_{\mathcal{I},\beta}(\mathcal{U}(x,\theta,(\alpha\cup\gamma)))(\nu) = \begin{cases} \rho_{\mathcal{I},\beta}(\mathcal{U}(x,\theta,\alpha)\cup\mathcal{U}(x,\theta,\gamma))(\nu) & \text{if } u(\alpha) \text{ and } u(\gamma) \\ \rho_{\mathcal{I},\beta}(x:=\theta;(\alpha\cup\gamma))(\nu) & \textit{otherwise} \end{cases}$$

$$\rho_{\mathcal{I},\beta}(\mathcal{U}(x,\theta,(\alpha^*)))(\nu) = \begin{cases} \rho_{\mathcal{I},\beta}(\mathcal{U}(x,\theta,\alpha)^*)(\nu) & \text{if } u(\alpha) \\ \rho_{\mathcal{I},\beta}(x:=\theta;\alpha^*)(\nu) & \textit{otherwise} \end{cases}$$

For the sequential composition we have to distinguish two cases.

- The first case is that the assignment is applicable on α as well as β. In this case we know that neither of these programs alter the valuation of x. Therefore we can conclude from the knowledge that elementary programs are evaluated equally that in this case the sequential composition is evaluated the same in ν and μ.

$$\begin{aligned}\rho_{\jmath,\beta}(\mathcal{U}(x,\theta,\alpha);\ \mathcal{U}(x,\theta,\gamma))(\nu) &\overset{(2.2)}{=} \rho_{\jmath,\beta}(\alpha;\ \gamma)(\mu)\\ &= \rho_{\jmath,\beta}(x := \theta;\ \alpha;\ \gamma)(\nu)\end{aligned}$$

- For the other case that the assignment is not applicable a program fragment is inserted that performs a state transition to μ. Therefore the resulting formula is directly what we have to show:

$$\rho_{\jmath,\beta}(x := \theta;\ \alpha;\ \gamma)(\nu)$$

The same arguments lead to the conclusion that the non-deterministic choice as well as the non-deterministic repetition are eveluated equally.

□

The next proposition to proof is proposition 3.

Proof. From the definition of the application of assignments (definition 32) we can derive the following cases for the application on formulas:

$$\begin{aligned}& val_{\jmath,\beta}(\nu,\mathcal{U}(x,\theta,p(t_1,\ldots,t_n)))\\ =\ & val_{\jmath,\beta}(\nu,p(\mathcal{U}(x,\theta,t_1),\ldots,\mathcal{U}(x,\theta,t_n)))\\ =\ & \jmath(p)(val_{\jmath,\beta}(\nu,\mathcal{U}(x,\theta,t_1)),\ldots,val_{\jmath,\beta}(\nu,\mathcal{U}(x,\theta,t_n)))\\ \overset{(2.1)}{=}\ & \jmath(p)(val_{\jmath,\beta}(\mu,t_1),\ldots,val_{\jmath,\beta}(\mu,\mathcal{U}(x,\theta,t_n)))\\ =\ & val_{\jmath,\beta}(\mu,p(t_1,\ldots,t_n))\end{aligned}$$

For this elementary formula the equality of the valuation follows from the equallity of the valuation of terms.

$$\begin{aligned}val_{\jmath,\beta}(\nu,\mathcal{U}(x,\theta,(\neg\varphi))) &= val_{\jmath,\beta}(\nu,\neg(\mathcal{U}(x,\theta,\varphi)))\\ &\overset{(2.3)}{=} val_{\jmath,\beta}(\mu,\neg(\varphi))\\ &= val_{\jmath,\beta}(\nu,[x := \theta]\neg(\varphi))\end{aligned}$$

$$\begin{aligned}
val_{\jmath,\beta}(\nu,\mathcal{U}(x,\theta,(\varphi\ op\ \psi))) &= val_{\jmath,\beta}(\nu,(\mathcal{U}(x,\theta,\varphi)\ op\,\mathcal{U}(x,\theta,\psi))) \\
&\overset{(2.3)}{=} val_{\jmath,\beta}(\mu,(\varphi\ op\ \psi)) \\
&= val_{\jmath,\beta}(\nu,[x:=\theta](\varphi\ op\ \psi)) \\
&\quad \text{for } op\in\{\wedge,\vee,\rightarrow,\leftrightarrow\} \\
val_{\jmath,\beta}(\nu,\mathcal{U}(x,\theta,(\forall Y\varphi))) &= val_{\jmath,\beta}(\nu,\forall Y(\mathcal{U}(x,\theta,\varphi))) \\
&\overset{(2.3)}{=} val_{\jmath,\beta}(\mu,\forall Y(\varphi)) \\
&= val_{\jmath,\beta}(\nu,[x:=\theta]\forall Y(\varphi)) \\
val_{\jmath,\beta}(\nu,\mathcal{U}(x,\theta,(\exists Y\varphi))) &= val_{\jmath,\beta}(\nu,\exists Y(\mathcal{U}(x,\theta,\varphi))) \\
&\overset{(2.3)}{=} val_{\jmath,\beta}(\mu,\exists Y(\varphi)) \\
&= val_{\jmath,\beta}(\nu,[x:=\theta]\exists Y(\varphi))
\end{aligned}$$

The first three cases do not alter the state thus we can conclude that they are evaluated equally. This is also obvious in the cases of quantified formulas.

$$\begin{aligned}
val_{\jmath,\beta}(\nu,\mathcal{U}(x,\theta,([\alpha]\varphi))) &= \begin{cases} val_{\jmath,\beta}(\nu,[\mathcal{U}(x,\theta,\alpha)](\mathcal{U}(x,\theta,\varphi))) & \text{if } u(\alpha) \\ val_{\jmath,\beta}(\nu,[x:=\theta][\alpha]\varphi) & otherwise \end{cases} \\
val_{\jmath,\beta}(\nu,\mathcal{U}(x,\theta,(\langle\alpha\rangle\,\varphi))) &= \begin{cases} val_{\jmath,\beta}(\nu,\langle\mathcal{U}(x,\theta,\alpha)\rangle\,(\mathcal{U}(x,\theta,\varphi))) & \text{if } u(\alpha) \\ val_{\jmath,\beta}(\nu,[x:=\theta]\,\langle\alpha\rangle\,\varphi) & otherwise \end{cases}
\end{aligned}$$

The cases for formulas that are prefixed by a modality split in two cases again. In the first case the valuation of x is not changed within the execution of the hybrid program α, thus we can conclude that it is evaluated the same in ν and μ. For the case that α does modify the valuation of x, a state transition to μ is inserted, thus the program α is executed always starting from a state μ and evaluation of the complete formula is identical for assignment case and the assignment application case.

□

Soundness of rule R25. The rule R25 is defined as:

$$\text{(R25)}\quad \frac{\forall T[x:=T]\varphi}{[x:=*]\,\varphi}$$

We want to show that it is locally sound. Therefore we need to proof that for all valuations of free variables β and all states ν if the premisses of the rule are valid under all interpretations $\jmath$ so are its conclusions. We have to show that

$$\forall T[x:=T]\varphi \models_{\beta,\nu} [x:=*]\,\varphi$$

holds. Using definition 36 we get:

$$\forall\beta\ \forall\nu\ (\forall\jmath\ \ (\jmath,\beta,\nu\models\forall T[x:=T]\varphi)\ \Rightarrow\ \forall\jmath\ \ (\jmath,\beta,\nu\models[x:=*]\,\varphi))$$

A implication can only be false, if the premisses are true and the conclusions are false. Assuming that the premisses are true, we can conclude that for a state ν $[x := T]\varphi$ holds for every value of T we choose.

The definition of the semantics of the random assignment (definition 26) says that every successor state of the random assignment is equal to the current state, but the value of the randomly assigned variable is an arbitrary real number. Assuming that there is a valid successor state μ in which φ becomes false.

By assumption the formula $[x := T]\varphi$ holds for every value of T in the state μ as well, as it only differs from ν in the value of x which does is not used to evaluate $[x := T]\varphi$ as the interpretation of x is changed to T. This is a contradiction as φ becomes false if we choose the value $\mu(x)$ for x and which means $[x := T]\varphi$ becomes false if we choose this value for T. □

The proof for R26 can be performed along the lines of this proof.

Soundness of R29. We have to proof that the rule R29 is sound. This can be done similar to the proof of the invariant rule in [Pla04]. The definition of the rule is:

$$\text{(R29)} \quad \frac{\Gamma \vdash \mathcal{A}p, \Delta \qquad p \vdash [\alpha]\, p \qquad p \vdash \varphi}{\Gamma \vdash \mathcal{A}\, [\alpha^*]\, \varphi, \Delta}$$

For proving that this rule is sound, we have to show that

$$(\Gamma \vdash \langle\!\langle \mathcal{A} \rangle\!\rangle\, p, \Delta) \land (p \vdash [\alpha]\, p) \land (p \vdash \varphi) \models \Gamma \vdash \langle\!\langle \mathcal{A} \rangle\!\rangle\, [\alpha^*]\, \varphi, \Delta$$

With definition 28 (definition of sequents) we can infer:

$$\psi := (\Gamma \to (\langle\!\langle \mathcal{A} \rangle\!\rangle\, p \lor \Delta)) \land (p \to [\alpha]\, p) \land (p \to \varphi) \models \Gamma \to \langle\!\langle \mathcal{A} \rangle\!\rangle\, [\alpha^*]\, \varphi \lor \Delta$$

We proof this inductively over the number of iterations performed by the loop. Assume that the loop is performed exactly n times.

IH If the transition (ν, μ) with p being valid in ν is performed by α^* within a maximum of n iterations, then ψ holds.

IA For the case $n = 0$, we can conclude from $p \to \varphi$ that ψ is valid.

IS For the case $n > 0$, we know, by assumption, that p holds in the state ν. Let ν' be some successor state with $(\nu, \nu') \in \rho_{\mathcal{I},\beta}(\alpha)$. We know by premisses that p holds in ν'. Assuming that the loop will be terminated within less than n iterations from the state ν', we can conclude using IH that $[\alpha^*]\, \varphi$ holds in ν'. This means that $[\alpha^*]\, \varphi$ also holds in ν, as ν' is the successor of ν by performing α once.

For an arbitrary state μ that satisfies the premisses

$$(\Gamma \to (\langle\!\langle \mathcal{A} \rangle\!\rangle\, p \lor \Delta)) \land (p \to [\alpha]\, p) \land (p \to \varphi) \models \Gamma$$

we can distinguish two cases. Either $\Gamma \to \Delta$ is true, which means that μ also models $\Gamma \to \langle\!\langle \mathcal{A} \rangle\!\rangle [\alpha^*] \varphi \vee \Delta$. Or Γ is true but Δ is false. In this case we know that $\langle\!\langle \mathcal{A} \rangle\!\rangle p$ holds in μ. Further, we can conclude that there is a state ν with $(\nu, \mu) \in \rho_{\jmath,\beta}(\mathcal{A})$ and p being valid in ν, which, as the induction shows, implies that also $[\alpha^*] \varphi$ is true in ν. This leads to the conclusion that $\langle\!\langle \mathcal{A} \rangle\!\rangle [\alpha^*] \varphi$ holds in μ as well. □

Soundness of R12. The definition of rule R12 is

$$\text{(R12)} \quad \frac{\Gamma \vdash \varphi[X \mapsto sk_X(Y_1, \ldots, Y_n)], \Delta}{\Gamma \vdash \forall X \varphi, \Delta}$$
$$\text{where } sk_X \text{ new, } Y_i \text{ free in } \varphi$$

For proving the local soundness of R12 we have to show, that for all valuations of free variables β and all states ν if the premisses of the rule are valid under all interpretations $\jmath$ so are its conclusions.

$$\vdash \varphi(sk_X(Y_1, \ldots, Y_n) \models_{\beta,\nu} \vdash \forall X \varphi(X)$$

Using definition 36 and the definition of the sequent we get:

$$\forall \beta \ \forall \nu \ (\forall \jmath \ \ (\jmath, \beta, \nu \models \varphi(sk_X(Y_1, \ldots, Y_n)) \Rightarrow \forall \jmath \ \ (\jmath, \beta, \nu \models \forall X \varphi(X)))$$

We assume $\jmath, \beta, \nu \not\models \forall X \varphi(X)$, i.e. there is an element $c \in \mathbb{R}$ such that $\jmath, \beta[X \mapsto c], \nu \models \varphi(X)$. This means that there is an interpretation for sk_X, the one that always maps to c, such that $\jmath, \beta, \nu \not\models \varphi(sk_X(Y_1, \ldots, Y_n)$. □

Soundness of R14. The definition of rule R14 is

$$\text{(R14)} \quad \frac{\Gamma \vdash \Delta, qE(\forall X(\Gamma' \vdash \Delta'))}{\Gamma, \Gamma'[X \mapsto sk_Y(\ldots)] \vdash \Delta'[X \mapsto sk_Y(\ldots)], \Delta}$$

For proving the local soundness of R14 we have to show that

$$\vdash qE(\forall X(\Gamma' \vdash \Delta') \models_{\beta,\nu} \Gamma'[X \mapsto sk_Y(\ldots)] \vdash \Delta'[X \mapsto sk_Y(\ldots)]$$

holds. Using definition 36 we get:

$$\forall \beta \ \forall \nu \ (\forall \jmath \ \ (\jmath, \beta, \nu \models \vdash qE(\forall X(\Gamma' \vdash \Delta')) \Rightarrow$$
$$\forall \jmath \ \ (\jmath, \beta, \nu \models \Gamma'[X \mapsto sk_Y(\ldots)] \vdash \Delta'[X \mapsto sk_Y(\ldots)]))$$

We assume $\jmath, \beta, \nu \not\models \Gamma'[X \mapsto sk_Y(\ldots)] \vdash \Delta'[X \mapsto sk_Y(\ldots)]$, i.e. there is at least one interpretation for sk_Y such that $\Gamma'[X \mapsto sk_Y(\ldots)] \vdash \Delta'[X \mapsto sk_Y(\ldots)]$ becomes false. The function sk_Y is a Skolemfunction, so it is rigid as well as its parameters. This rigidness makes it possible to extract the value of sk_Y from this interpretation and name it c. This value c is an counter example for $\forall X(\Gamma' \vdash \Delta')$. As the quantifier elimination is an equivalence transformation on first-order formulas that $\jmath, \beta, \nu \not\models \vdash qE(\forall X(\Gamma' \vdash \Delta')$. This means that

$$\vdash qE(\forall X(\Gamma' \vdash \Delta') \models_{\beta,\nu} \Gamma'[X \mapsto sk_Y(\ldots)] \vdash \Delta'[X \mapsto sk_Y(\ldots)]$$

holds. □

Soundness of R15. The definition of rule R15 is

$$\text{(R15)} \quad \frac{qE(\exists X \bigwedge_i (\Gamma_i \vdash \Delta_i))}{\Gamma \vdash \Delta, \exists X \varphi}$$

For proving the soundness of R15 we have to show that:

$$\vdash qE(\exists X(\bigwedge_i \Gamma_i \vdash \Delta_i)) \models_{\beta,\nu} \Gamma \vdash \exists X \varphi, \Delta$$

$$\forall \beta \ \forall \nu \ (\forall \jmath \ \ (\jmath, \beta, \nu \models \vdash qE(\exists X(\bigwedge_i \Gamma_i \vdash \Delta_i))) \Rightarrow \forall \jmath \ \ (\jmath, \beta, \nu \models \Gamma \vdash \exists X \varphi, \Delta))$$

As the quantifier elimination is an equivalence transformation we can infer:

$$\forall \beta \ \forall \nu \ (\forall \jmath \ \ (\jmath, \beta, \nu \models \vdash \exists X(\bigwedge_i \Gamma_i \vdash \Delta_i)) \Rightarrow \forall \jmath \ \ (\jmath, \beta, \nu \models \Gamma \vdash \exists X \varphi, \Delta))$$

Using the definition of the existence quantifier (definition 27.8) and the fact that X does not occur free in $\Gamma \vdash \exists X \varphi, \Delta$ we get:

$$\forall \beta \ \forall \nu \ (\forall \jmath \ \ (\jmath, \beta[X \mapsto c], \nu \models \vdash (\bigwedge_i \Gamma_i \vdash \Delta_i)) \Rightarrow \forall \jmath \ \ (\jmath, \beta[X \mapsto c], \nu \models \Gamma \vdash \exists X \varphi, \Delta))$$

Since the side deduction triggered by R15 is locally sound, we can conclude:

$$\forall \beta \ \forall \nu \ (\forall \jmath \ \ (\jmath, \beta[X \mapsto c], \nu \models \vdash (\Gamma \vdash \Delta, \varphi)) \Rightarrow \forall \jmath \ \ (\jmath, \beta[X \mapsto c], \nu \models \Gamma \vdash \exists X \varphi, \Delta))$$

Again we use definition 27.8:

$$\forall \beta \ \forall \nu \ (\forall \jmath \ \ (\jmath, \beta, \nu \models \vdash \exists X(\Gamma \vdash \Delta, \varphi)) \Rightarrow \forall \jmath \ \ (\jmath, \beta, \nu \models \Gamma \vdash \exists X \varphi, \Delta))$$

Using that X does not occur neither in Γ nor in Δ we get:

$$\forall \beta \ \forall \nu \ (\forall \jmath \ \ (\jmath, \beta, \nu \models \vdash \Gamma \vdash \Delta, \exists X \varphi) \Rightarrow \forall \jmath \ \ (\jmath, \beta, \nu \models \Gamma \vdash \exists X \varphi, \Delta))$$

□

The proofs for the soundness of the rules R13 and R16 can be performed along the lines of the provided proofs for the rules R12 and R15.

Chapter 3

Design

3.1 Overview

To provide a computer aided proof system, we have chosen a two tier architecture. On the one hand we have KeY as the user interface and for providing deductive methods and on the other hand we have an interface to an arithmetic solver. For a first implementation we intend to integrate KeY version 1.0 with Mathematica 5. The user can perform the proof in KeY and can invoke an arithmetic solver using calculus rules. We have chosen Mathematica as it is known for its symbolic methods and provides a well documented JAVA interface called J/Link [Wol07].

In this chapter we describe the design of our software system as well as the necessary part of the architecture of KeY. The structure of this chapter is as follows.

First we describe the architecture of the KeY system and which components are changed to integrate support for $\mathsf{d}\mathcal{L}$. Afterwards we will describe how calculus rules are embedded into KeY.

In the second section we will discuss two alternative approaches to extend KeY such that it can handle $\mathsf{d}\mathcal{L}$ formulas. The first is mapping the $\mathsf{d}\mathcal{L}$ syntax to the syntax of the already supported JAVACARD DL. The second alternative is extending the parsers and data structures to handle a new syntax for the $\mathsf{d}\mathcal{L}$ formulas. We decided that the latter alternative is the promising approach and present the necessary design steps for implementing it.

Another challenge is the integration of KeY with arithmetic solvers. The last section of this chapter covers this point and describes our approach to integrate KeY with Mathematica.

3.1.1 Architecture of the KeY Prover

In figure 3.1 a schematic view on the architecture of KeY [BHS07] is provided. The user communicates from two sides with the system. First he provides the *Problem File* containing the system specification as well as the property to check. Second he uses the GUI to perform the proof. The problem file is processed by the problem parser which generates `Namespaces` containing all declared variables, functions, predicates and sorts. It also translates the system specification and the problem into `Term` objects. This information is used to initialize the proof. The

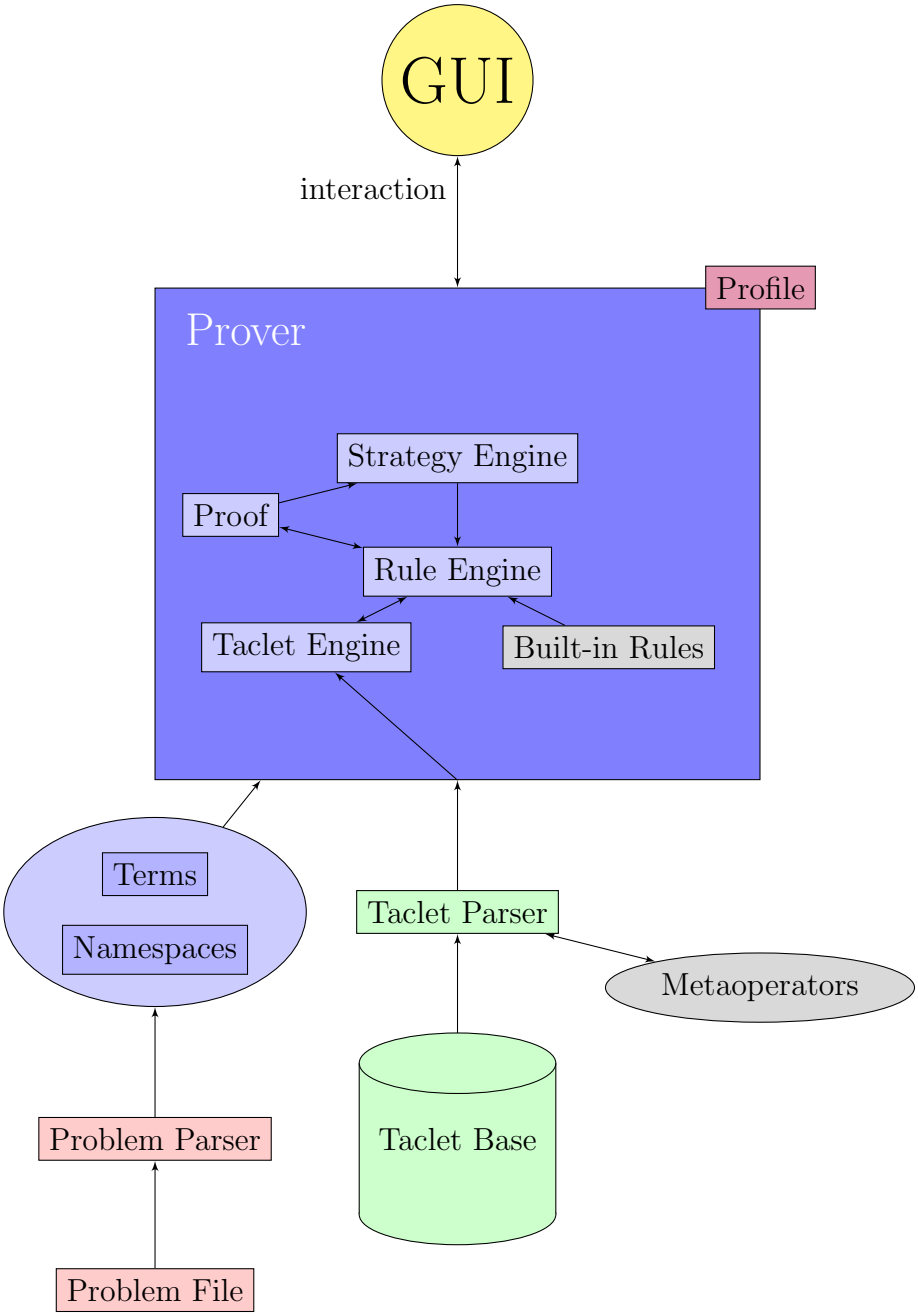


Figure 3.1: KeY Architecture

specified problem is the root node of the proof tree. `Term` objects are the common representation of formulas inside KeY.

The `Term` data structures are immutable for two reasons. First, the performance of the matching algorithms is significantly higher, as equality tests can be done by comparing memory references. Second, no rule can, by programming mistake, alter a node of the proof graph, it is only possible to append or discard nodes. To assert that for one logic object the same JAVA object is used, they are cached in the `Namespaces`.

For the initialization also a so called `Profile` is used. It specifies from which file the taclets are loaded and which built-in rules to use. Taclets are a special representation for calculus rules. They were introduced to make it easier to add or change rules, as well as to avoiding programming mistakes in the implementation of rules. Rules formulated in the taclet language are easier to check for soundness as rules programmed in pure JAVA as the taclet language is simpler and specially designed for the specification of calculus rules. For a detailed description of taclets and built-in rules see section 3.1.2. By default KeY supplies a `Profile` for pure FOL as well as one for JAVA CARD DL. We add a `Profile` especially for $\mathtt{d}\mathcal{L}$.

The taclets are loaded using a specific *Taclet Parser*. This parser is an extended version of the problem parser. It can handle schematic problem descriptions which can contain calls to metaoperators as well as schema variables.

The *Taclet Engine* is an abstraction of the taclet processing mechanism. It can check whether a taclet is applicable on a specific formula and, if this is the case, apply it to that formula. The *Rule Engine* is the combination of the taclet engine and the built-in rules. It uses the taclet engine to handle taclet based rules and can process the built-in rules as well. The *Strategy Engine* is used for automation of proofs. It processes a proof strategy defined by the `Profile`. A strategy is basically a cost relation. It maps costs, depending on the current formula and the previous rules applied, to rules. The *cheapest* applicable rule is to be applied. The strategy engine can determine as well which proof goal should be processed next.

For our extension we leave the GUI unchanged. Also we did not alter the strategy engine, nor the taclet engine or the rule engine. We created a taclet base for the $\mathtt{d}\mathcal{L}$ calculus based on the existing taclet base for propositional logic. Also we extended the parsers thus they can understand $\mathtt{d}\mathcal{L}$ formulas. Additionally, we added built-in rules that are only necessary in the $\mathtt{d}\mathcal{L}$ context. They are used to handle the real valued arithmetic that is inherent to $\mathtt{d}\mathcal{L}$ but does not occur in the JAVACARD DL context, as JAVA only uses integer arithmetics. To make it possible to translate most of the rules into the taclet language we added a few new metaoperators.

3.1.2 Calculus Embedding

In KeY there are two ways to implement the calculus rules. They can either be described as a taclet or they can be implemented in pure JAVA as so called built-in rule. In this section we will describe both alternatives and distinguish the cases where to use which alternative.

Taclets KeY features a taclet language [BHS07] to describe calculus rules. An example taclet is shown in figure 3.2.

— Taclet —

```
all_right {
    \find (==> \forall u; b)
    \varcond (\new(sk, \dependingOn(b)))
    \replacewith (==> {\subst u; sk}b)
    \heuristics (delta)
};
```

— Taclet —

Figure 3.2: Example for a taclet

We will describe only the relevant part of the taclet language here. A complete specification can be found in [BHS07].

Taclets consist of a `find` clause, to ascertain if they are applicable and to determine which formula in a sequent is changed as well as a `replacewith` clause to describe their effect. The rule in the example watches on an universal quantifier on the right side of a sequent. The quantifier is dropped and a new Skolem symbol is added substituting the prior quantified variable. The Skolem symbol is introduced using a variable condition, specified with the keyword `varcond`. In this case two condition are made for the Skolem symbol. First it has to be a fresh variable, i.e. must not occur anywhere else in the proof, and a dependency to the free variables occurring in the quantified formula is added.

The keyword `varcond` is always used to specify conditions for symbols that are introduced by the taclet. In the example two conditions on `sk` are specified. The keyword `new` donates that `sk` is a fresh symbol, whereas `dependingOn` is used to specify that it is depending on the free variables occurring in the formula `b`.

There is another keyword called `assumes` that can be used to specify that for making the rule applicable a certain formula has to occur in the sequent. This is illustrated in figure 3.3. Formulas specified in the `find` clause are replaced by those specified in the `replacewith` clauses. The formulas specified in the `assume` clause are left unchanged.

Also a heuristic group can be specified using the keyword `heuristics`. The heuristic group is used when the prover is in the automatic mode. The taclets are sorted into groups, to make it easier to describe which taclets should be applied with which priority.

The formulas specified in the different clauses are basically the same that can occur in the problem description. Additionally, they can contain so called *schema variables*. These schema variables are used as place holders for a special sort of formula element. While checking if a rule is applicable, the existing schema variables are instantiated with terms contained in the current formula. The rule is applicable, if all schema variables can be instantiated correctly, i.e. the instance is

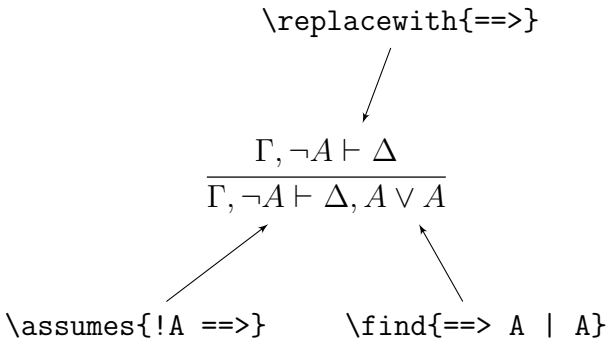


Figure 3.3: Relation between sequent calculus rules and taclets

chosen with respect to the type of the schema variable, and the structure matches the formula structure specified in the taclet. In our example there are the schema variables `u`, `b` and `sk`. The schema variable `u` is a placeholder for an arbitrary logical variable, `b` is schema of a formula and `sk` is a placeholder for a function symbol.

Another common sort for a schema variable is e.g. `\modaloperator{ diamond, box }`. A schema variable with this type matches the modalities box (`[]`) and diamond (`<>`) and thus can be used to easily model the calculus rules that do not depend on the type of the modality like R19.

There are two kinds of taclets. First there are so called rewrite taclets that can be applied to any formula or subformula. This can be used for e.g. replacing modalities that only contain assignments with updates. Updates are the approach used in KeY to delay the application of assignments. The second type of taclets is the straight forward implementation of rules from a sequent calculus. They contain the sequent symbol (`==>`) in the `find` as well as the `replacewith` clause, so they can handle the different sides of the sequent differently. Those taclets only work on complete formulas, whereas rewrite taclets can match parts of formulas.

Metaoperators are special symbols that make it possible to express complex transformations. These metaoperators are implemented in pure JAVA, thus they can be used for e.g. the handling of differential equation systems. These systems can be matched using the taclet mechanism but for solving them we need to call an arithmetic solver which can only be done using pure JAVA code.

While applying the taclet the metaoperator is evaluated. The call to the metaoperator is replaced by its output.

An example for a taclet using a metaoperator is presented in figure 3.4. Here the metaoperator `#dlunwind` is used. The metaoperator transforms its input depending on the modality type as demanded by the rules R23 or R22. In this example `#dl` is a schema variable that can stand for an arbitrary hybrid program.

Even with the use of metaoperators, not every sound rule of a sequent calculus can be expressed using taclets. Taclets can only be used to express rules that are sufficiently local, i.e. it is for example not possible to express a rule that explicitly refers to all of the formulas in the sequent.

— Taclet —

```
loop_unwind {
        \find (
                \modality{#allmodal}
                        #dl*
                \endmodality(post)
        )
        \replacewith(
                #dlunwind(
                        \modality{#allmodal}
                                #dl
                        \endmodality(post)
                )
        )
};
```

— Taclet —

Figure 3.4: Example for a taclet using a metaoperator

A solution to implement those rules are so called built-in rules.

Built-in Rules The second way to implement calculus rules in KeY are so called built-in rules. These rules can work on the whole sequent or even on the whole proof. They are written in pure Java, thus implementation faults can easily occur in these implementations. For this reason, they should only be used if its impossible to express the rule as taclet, as the restrictions made by the taclet language help to avoid implementation faults.

All built-in rules have to implement the interface `BuiltInRule` which is a subinterface of `Rule`. The interface `Rule` demands that the implementing class implements a method for the application of the rule. This method is called `apply`. It takes 3 parameters. The first is the current goal, the second is the current configuration represented as `Services` object and the third is a `RuleApp` object that describes to which formula the rule is applied. The last object can be extended to pass more informations to the built-in rule. This function has to return a `ListOfGoal` which is the list of successor nodes in the tree. If this function returns `null` the rule does not have any effect on the proof. An empty list as return value stands for closing the current goal. The Interface `BuiltInRule` also demands that a method called `isApplicable` is implemented. This method is used to determine whether the built-in rule is applicable or not.

One example for such a case is the `ReduceSequence` rule, that is used to eliminate arithmetic using the whole knowledge provided by a sequent. When applying this rule the user may provide a list of variables that should be reduced. This list is passed to the rule using an extended `RuleApp` object. The rule is described

more extensively in section 4.2.2.2.

3.2 Integration

KeY is designed to prove JAVA CARD DL formulas, thus the parsers only accept JAVA programs in the modalities. Since changing this behavior is time intensive, we have considered a JAVA embedding of the hybrid programs.

3.2.1 Java Embedding of Hybrid Programs

In this section we describe our approach to rewrite hybrid programs in a JAVA syntax and argue why we decided to implement our own parser and program representation instead of embedding the hybrid programs into JAVA syntax.

The chop operator can be mapped straight forward to the sequential composition used in JAVA, as it already uses the `;` as representation. We intended to represent the choice $(\alpha \cup \gamma)$ using a construct like `if(nondet()) alpha else gamma`. The condition `nondet()` has to be a function of type boolean that is handled specially by the rules, i.e. there have to be rules that can handle this special symbol, as well as it has to be prevented that other rules handle this symbol like a common function symbol.

The symbol `nondet()` is used to express non-determinism, i.e. it can randomly change its value. The same symbol is used to represent the non-deterministic repetition (α^*) as `while(nondet()) alpha`. To integrate state assertion we intended to use `if` as well. One major problem with this is the fact that KeY has to distinguish between normal predicate symbols and the special symbol `nondet()`. This could be achieved by introducing a new schema variable type that matches all expressions others than `nondet()`.

Also a formula representation has to be found for quantifier free formulas within JAVA. It is necessary that the user defines every variable used inside the program with a JAVA type. Also JAVA only provides boolean operators for the negation, disjunction and conjunction. Thus the implication and equivalence can only be expressed as special function symbols.

So we have a representation for the combination operators and the state assertions, but what about the assignment and the continuous evolutions?

The translation of assignments is straight forward again. JAVA uses assignments as well, so we can just keep its syntax for that. The assignment $(x := \theta)$ would simply become `x = theta`. The real problem is the embedding of the continuous evolutions. We considered representing them within a `do { ... } while(diffsystem);` statement and use a boolean array as encapsulation of the system itself. The representation of a differential equation system $(\{\dot{x} = \theta_1, \dot{y} = \theta_2, x < \theta_3\})$ is shown in figure 3.5.

The fact that using JAVA constructs for the syntax of hybrid programs would lead to unreadable formulas, especially of those containing differential equation systems, has motivated the decision to extend KeY in order that it can handle the syntax of hybrid programs. This strategy is corroborated by the fact

— JAVA —
```
do {
        new Boolean[] {
                dot(x) == theta1,
                dot(y) == theta2,
                x < theta3
        };
} while(diffsystem);
```
— JAVA —

Figure 3.5: JAVA embedded differential equation system

that hybrid programs behave differently in comparison to JAVA. In JAVA, statements can have side effects, which is a compelling case to handle statements in, e.g., a `while`-loop condition carefully. An illustrating example would be `while(i++ < 20) { ... }`. Here the condition is not a simple condition, but it is a combination of an assignment and a condition. The embedded assignment is the increment operator `++` that increases the value of `i` after the expression is evaluated.

Also in hybrid programs real valued variables are used, whereas KeY only provides rules for handling integers, thus we cannot reuse many of the existing rules. A major problem would also be error tracing for the user. KeY uses Recoder [GHT07] to parse the JAVA programs. As alternating Recoder is unworthy of discussion, as even the KeY developers decided to rather implement an own JAVA parser for the schema part instead of extending Recoder, we can not prohibit function calls in expression and other JAVA syntax we cannot handle. These are convincing reasons to integrate a special syntax for hybrid programs.

3.2.2 Abstract Syntax of d$\mathcal{L}$ Formulas in KeY

To give an idea how the formulas used for system specification look like we present an abstract syntax description of the formulas, that can be specified in KeY in this section. We abstract from the use of parenthesis and abbreviate some productions. Table 3.1 shows the grammar of the abstract syntax. This syntax description is more technical than the syntax presented in section 2.2. Here we do not abstract from different objects representing formula parts that look identical. The abstract syntax should make it easier to understand our data representation presented in the following sections.

The connection between the syntax definition in section 2.2 and the abstract syntax presented here is as follows. The set of terms $\mathcal{T}(\mathcal{V}, \mathcal{S}, \Sigma)$ is constructed by the productions EXPR for terms in the FOL part of the formula and α-EXPR for terms used inside programs. As derivatives may only occur inside differential equation systems, a production is added that construct these terms. It is called DIFFEXPR. The set of hybrid programs is constructed by the production α for

FORM	::=	$\forall$XFORM $\mid \exists$XFORM $\mid [\alpha]$FORM $\mid \langle\alpha\rangle$ FORM
		$\mid$ FORM $\wedge$ FORM $\mid$ FORM $\vee$ FORM $\mid$ FORM $\rightarrow$ FORM
		$\mid \neg$FORM $\mid$ ATOM
ATOM	::=	P(EXPR, ..., EXPR) $\mid$ P
EXPR	::=	F(EXPR, ..., EXPR) $\mid$ F $\mid$ Y $\mid$ NUM
α	::=	$\alpha;\alpha \mid \alpha \cup \alpha \mid \alpha^* \mid$ ATOMIC-α
ATOMIC-α	::=	$?\alpha$-FORM $\mid \{$DIFFSYSTEM$\} \mid$ Z $:= \alpha$-EXPR $\mid$ Z $:= *$
α-FORM	::=	α-FORM $\wedge$ α-FORM $\mid$ α-FORM $\vee$ α-FORM
		$\mid$ α-FORM $\rightarrow$ α-FORM $\mid$ α-FORM $\leftrightarrow$ α-FORM $\mid \neg\alpha$-FORM
		$\mid \alpha$-ATOM
α-ATOM	::=	P(α-EXPR, ..., α-EXPR) $\mid$ P $\mid$ α-EXPR $\sim$ α-EXPR
α-EXPR	::=	F(α-EXPR, ..., α-EXPR) $\mid$ F $\mid$ α-EXPR $\oplus$ α-EXPR
		$\mid -\alpha$-EXPR $\mid$ (α-EXPR)$^{(\alpha\text{-EXPR})}$ $\mid$ Y $\mid$ NUM
DIFFSYSTEM	::=	(DIFFEQUATION $\mid \alpha$-FORM)(, DIFFEQUATION $\mid \alpha$-FORM)*
DIFFEQUATION	::=	$\dot{\text{Z}}$ = DIFFEXPR
DIFFEXPR	::=	f(DIFFEXPR, ..., DIFFEXPR) $\mid$ F $\mid \dot{\text{Z}} \mid$ Y $\mid$ NUM
		$\mid$ DIFFEXPR $\oplus$ DIFFEXPR $\mid -$DIFFEXPR
		$\mid$ (DIFFEXPR)$^{(\alpha-\text{EXPR})}$

Table 3.1: Abstract syntax of $\mathsf{d}\mathcal{L}$ formulas

compound programs and ATOMIC-α for elementary programs. The set of formulas $\mathcal{F}(\mathcal{V}, \mathcal{S}, \Sigma)$ is constructed by the production FORM for compound formulas and ATOM for elementary ones. The quantifier free formulas that may occur inside hybrid programs on the other hand are constructed by the production α-FORM.

Some of the symbols used in the figure are placeholders. The symbol X stands for a logic variable, Z stands for a program variable, whereas Y stands for either a logic or program variable. Numbers are represented by NUM. Predicate and function symbols are donated by P and F. Binary relation for comparing terms are given by $\sim \in \{<, \leq, =, \geq, >\}$. The mathematical operators are abbreviated by $\oplus \in \{+, -, \times, \div\}$.

We distinguish between the productions EXPR and α-EXPR here because the first one is part of the input language of KeY by default, whereas the latter one is part of our extension. One might remark that there are neither the usual mathematical function symbols nor the common predicates predefined in the FOL part of the formula construction. This is because the design of the KeY parser is different from ours. For the FOL part the mathematic function are defined in the taclet files and for easier input and better readability the parser and the GUI have a mapping between the functions and a human readable representation. For example in the taclet files for $\mathsf{d}\mathcal{L}$ there is a function `R mul(R,R)` which is printed as the common symbol $\times$. It is possible to switch this pretty printing off in the KeY GUI. But for some reason the functions are saved in the pretty printed version, which leads to problem with e.g. with the exponential function, as the common operator for exponential function (`^`) is used as an internal parser symbol.

As for dℒ programs these predicate and function symbols are essential, we decided to integrate them directly into our grammar.

The differential equation systems are represented by a comma separated list of differential equations and invariant constraints. The comma symbolises a conjunction. The differential equations (DIFFEQUATION) are equations that can contain a special symbol dot. It donates the Newton notation of derivatives. For example $\dot{z}$ is the first derivative of z with respect to time. It may be noted that the extension from a single differential equation with invariant to a system of differential equations with invariants is sound (see [Pla07c]).

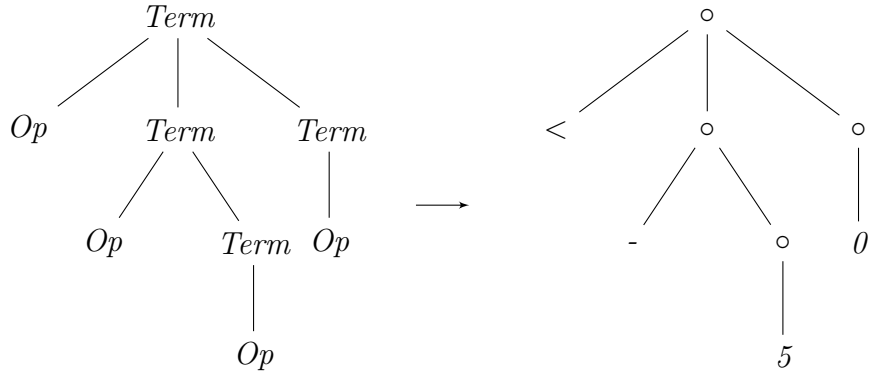
3.2.3 Syntax Tree

We represent the dℒ formulas using their abstract syntax tree produced by the abstract syntax described in the previous section. The representation of the first-order part of the formulas left unchanged to the prior implementation of KeY. In KeY there is a class called `Term`. A `Term` is an inner node of the syntax tree. The children of a `Term` are Operators, `JavaBlock` objects and `Term` objects again. One of the children of a `Term` is always an Operator. The operator implements an interface called `Operator`. The operators can be the common first-order functions, predicates or variables or operators like $\forall, \exists, \wedge, \vee, \ldots$, modalities, substitutions or updates. This covers the productions FORM, ATOM and EXPR.

If the operator is a modality, the `Term` has two other children. One is the `JavaBlock` containing the program description. The other child is postcondition of the modality. The `JavaBlock` itself contains a `Statement` which is the point we insert our data structures for hybrid programs. This statement is the root node of the data structures produced by the production α and its successors.

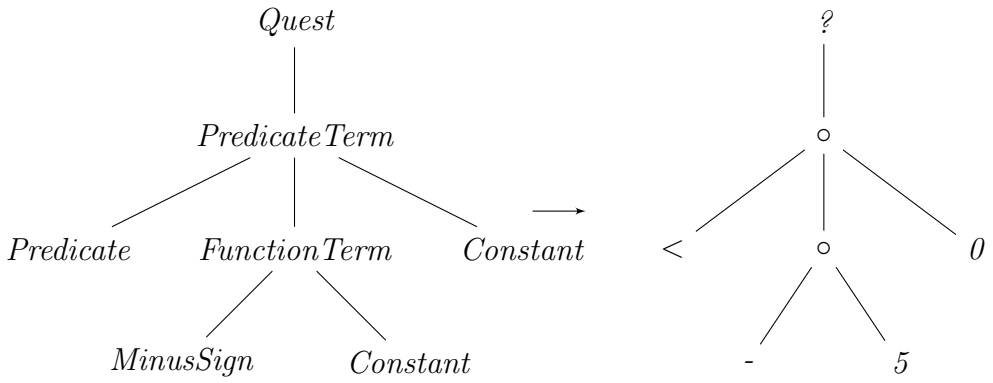
The following example illustrates how a simple formula is represented using the `Term` data structure.

Example 4. *The representation of the formula* $-5 < 0$ *would look like this:*



The data structures we use for hybrid programs are slightly different. The following example should illustrate the differences and a detailed description of the data structure design is provided in section 3.2.4.

Example 5. *The representation of the hybrid program* $?-5 < 0$ *looks as follows.*



3.2.4 Data Structures for Hybrid Programs

The architecture of KeY demands that all elements that can occur in a modality implement the interface `ProgramElement`. As the `Term` objects used for the formula representation of the FOL part in KeY do not implement this interface, it is impossible to use inside the programs. The methods used to access the data structures differ between the logic part and the program part and the `Term` class is not intended to implement the interface `ProgramElement`. Also the elements used for representation of the JAVA syntax contain JAVA specific data, that is not necessary in our context. For these reasons we decided to design a complete set of data structures for the hybrid programs.

In section 3.2.2 we presented the abstract syntax of the formulas. Section 3.2.3 was used to illustrate how the first-order part of the formulas is represented and where we integrate our extension. In this section we present the data structures used to represent the syntax for hybrid programs. We refer to the productions presented in table 3.1. Figure 3.6 shows the interface structure for the programs. These interfaces are used for representation of the results of the productions of α and ATOMIC-α. The figure 3.7 illustrates the interface structure for the formulas that can occur in the programs resulting from the productions α-FORM and α-ATOM. Figure 3.8 shows the interface structure for expressions that result from the productions α-EXPR, DIFFSYSTEM, DIFFEQUATION and DIFFEXPR.

The data structures for the hybrid programs are modelled with the abstract factory pattern [GHJV95]. For a description of this pattern see appendix A section A.3.1. This pattern is supported by the marker interface pattern [GHJV95] which is already embedded in the JAVA language (see section A.2.3). In our case our abstract factory is called `TermFactory`. The abstract products are the leafs of the interface hierarchy presented in this section.

To have a common supertype, we decided that all our interfaces specialize the interface `DLProgramElement`. This interface is a subinterface of `ProgramElement` so we satisfy the requirements of KeY for objects occurring in programs. The interface `ProgramElement` demands that the implementing classes provide two methods for the visitor pattern [GHJV95] (see section A.4.1).

We have decided that it is the most natural form for tree structures if the operator serves as the parent of its operands, thus we model the data structures in

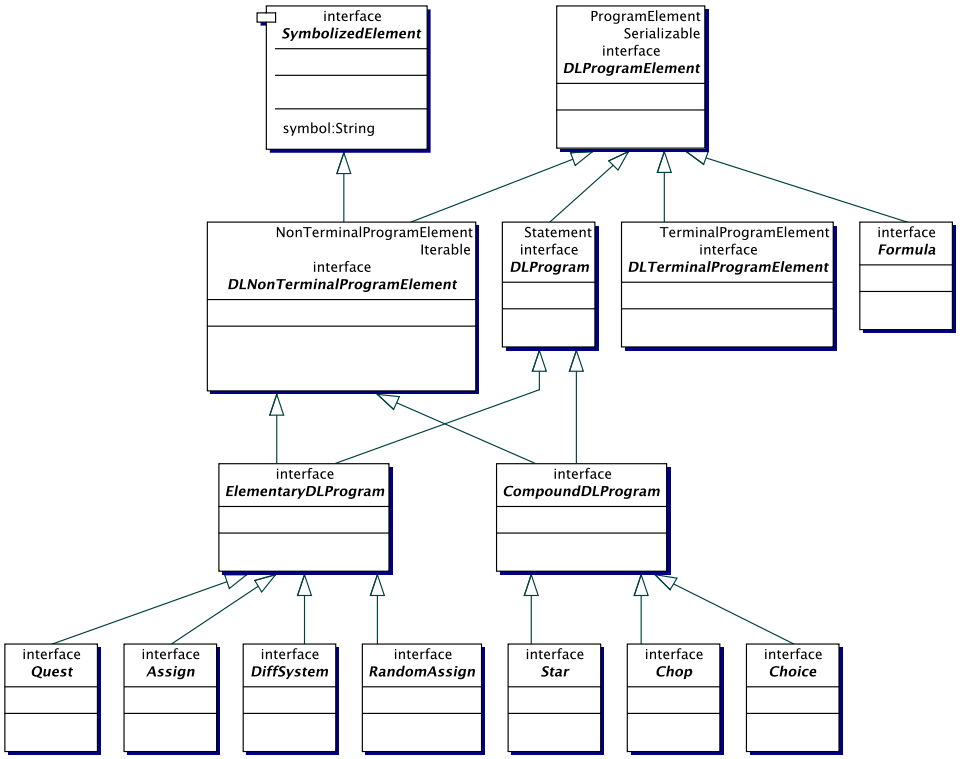


Figure 3.6: Data structures for $\mathsf{d}\mathcal{L}$ programs resulting from productions α and ATOMIC-α

this way where ever there is no disadvantage to the introduction of a composition object. This has no special advantage compared to the composition approach, but we had to make a decision and this way modelling looked more intuitive to us.

The non-terminal α is represented by the interface `DLProgram`. The interface `ElementaryDLProgram` represents the non-terminal ATOMIC-α. The operators on program level are `Chop` for the sequential composition, `Choice` for the non-deterministic choice and `Star` for the non-deterministic repetition. They all implement `DLNonTerminalProgramElement` and `DLProgram`. The interface `DLNonTerminalProgramElement` specifies that the implementing class can store children and these children can be accessed in the common way used in KeY inside programs. The `DLNonTerminalProgramElement` additionally demands that the iterator pattern is implemented. Is is especially useful in the context of JAVA 1.5 as it enables the usage of the extend `for`-loop.

The children of non-terminal `DLProgram` objects have the common type `DLProgram` again such that we can built up a recursive data structure. The leaves of the tree are `Quest` for a state assertion, `Assign` and `RandomAssign` for a (random) discrete state change as well as `DiffSystem` for continuous evolutions.

`DLProgram` implements the `Statement` interface supplied by KeY. As mentioned before the `JavaBlock` can store a `Statement` such that we can easily integrate it within the existing data structures for modalities by implementing the `Statement` interface.

Formulas are modelled to a large extend in the same way as programs. The formula operators all implement `DLNonTerminalProgramElement` and their children are formulas of type `Formula`. The interface `Formula` is the representation of the non-terminal α-FORM.

The first exception in the tree modelling are the data structures for formulas. Here we use a special element for composition of an operator and its operands (see example 5).

This way we need only one object per predicate name. They are cached using a `WeakHashMap`, thus they are freed when their name is not referenced anymore. This is important, as otherwise equality checks would be expensive. As the equality checks have to be performed to determine if a rule is applicable or not and these checks are performed many times it is necessary that these checks can be performed quickly. Two formulas are equal iff all their subterms at the same position are equal. A composition term is equal iff all its children are equal with respect to their order. As the predicates are shared, i.e. there is only one object per number, function or predicate symbol, we can use the reference equality (==) provided by JAVA to check this and delegation on the composition objects. Without the composition model, it would be necessary to determine equality of `String` objects. Another effect of this data structure is that we can share these predicates between proofs to save memory.

The composition operator on formula level is called `PredicateTerm` and its first child is always a `Predicate`, the rest is of type `Expression`. The interface `PredicateTerm` is the representation of the non-terminal α-ATOM.

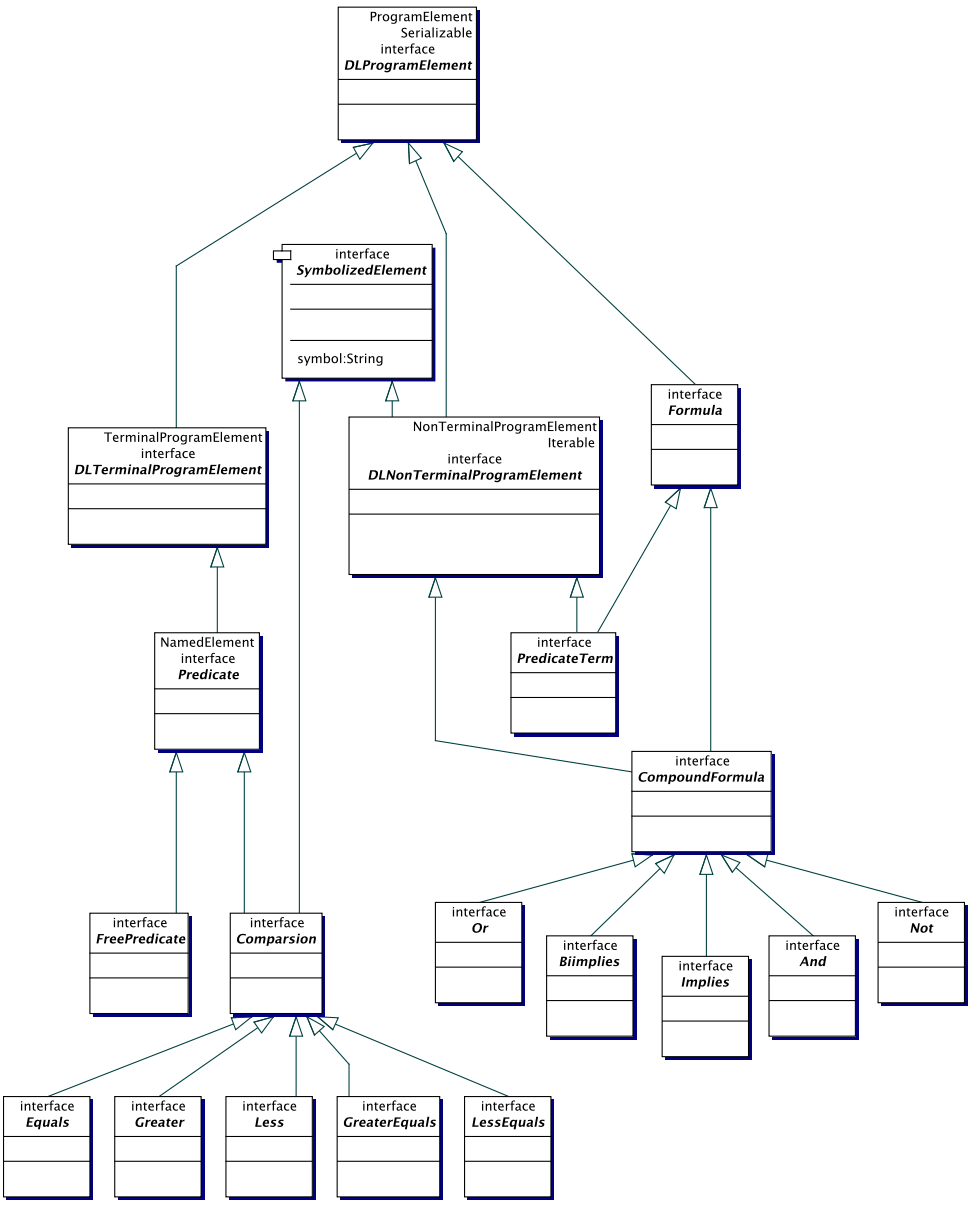


Figure 3.7: Data structures for formulas within $\mathsf{d}\mathcal{L}$ programs generated by the productions α-FORM and α-ATOM

The data structures for the expressions are shown in figure 3.8. A detailed view on the data structures for the functions is presented in figure 3.9. The expressions are related to the productions α-EXPR as well as DIFFEXPR.

For the same reasons we model expressions with a composition operator. Here it is called `FunctionTerm` and we share the functions instead of the predicates.

3.3 Mathematica Integration

Wolfram supplies a library for accessing Mathematica with JAVA functions. It is called J/Link [Wol07]. J/Link can start Mathematica kernels and pass over queries. These queries can either be `Strings` or of type `Expr`. We use the `Expr` representation as it supports our recursive algorithm and `String` related implementation faults can be avoided. The `Expr` format is a rather simple representation of formulas. Instead of using many different classes to represent the formula tree, `Expr` objects each have a node type. Only the leafs of the tree hold informations. The inner nodes are just composition nodes. The format has the expressive power of the `FullForm` strings in the Mathematica interface.

Our interface to Mathematica (see figure 3.10) is designed as a client-server architecture. There is a server which performs the calculations implemented in the class `KernelLinkWrapper` and on the client side there is the `IMathSolver` interface for accessing the server side functions. Client and server communicate using JAVA RMI.

RMI is the abbreviation for Remote Method Invocation. It is used for modelling distributed computation in JAVA. With RMI JAVA objects can easily be distributed between different applications on the same or on another machine. The function the client wants to access has to be declared in an interface in our case `IMathSolver`. By contract, all methods that can be accessed remotely have to have a `throws` clause that specifies that they may throw a `RemoteException`. Since JAVA 1.5 the stubs for the client side including the delegation code for the communication with the server is generated automatically. The parameters of the remote methods have to be serializable as they are passed as XML stream over the network. As not all objects used for the formula representation in KeY are serializable the translation is completely performed on the client side.

The `MathSolverManager` is the central class on client side. It provides access to the available arithmetic solvers. At the moment there is only an implementation for accessing Mathematica but the architecture is kept generic for later extensions. We use a XML [BPSM97] file for configuration. This file describes which solvers are available and how they have to be initiated.

The Mathematica interface on the client side is further separated into one wrapper class to implement the `IMathSolver` interface and a converter class called `MathematicaDLBridge`. The latter one uses different specialized converters. The translation from differential equation systems is done by the `DL2ExprConverter`. The `Term2ExprConverter` is capable of translating first-order formulas that are represented as `Term` objects into the `Expr` format. The third convert is the `Expr2TermConverter`. It translates the result provided by Mathematica into

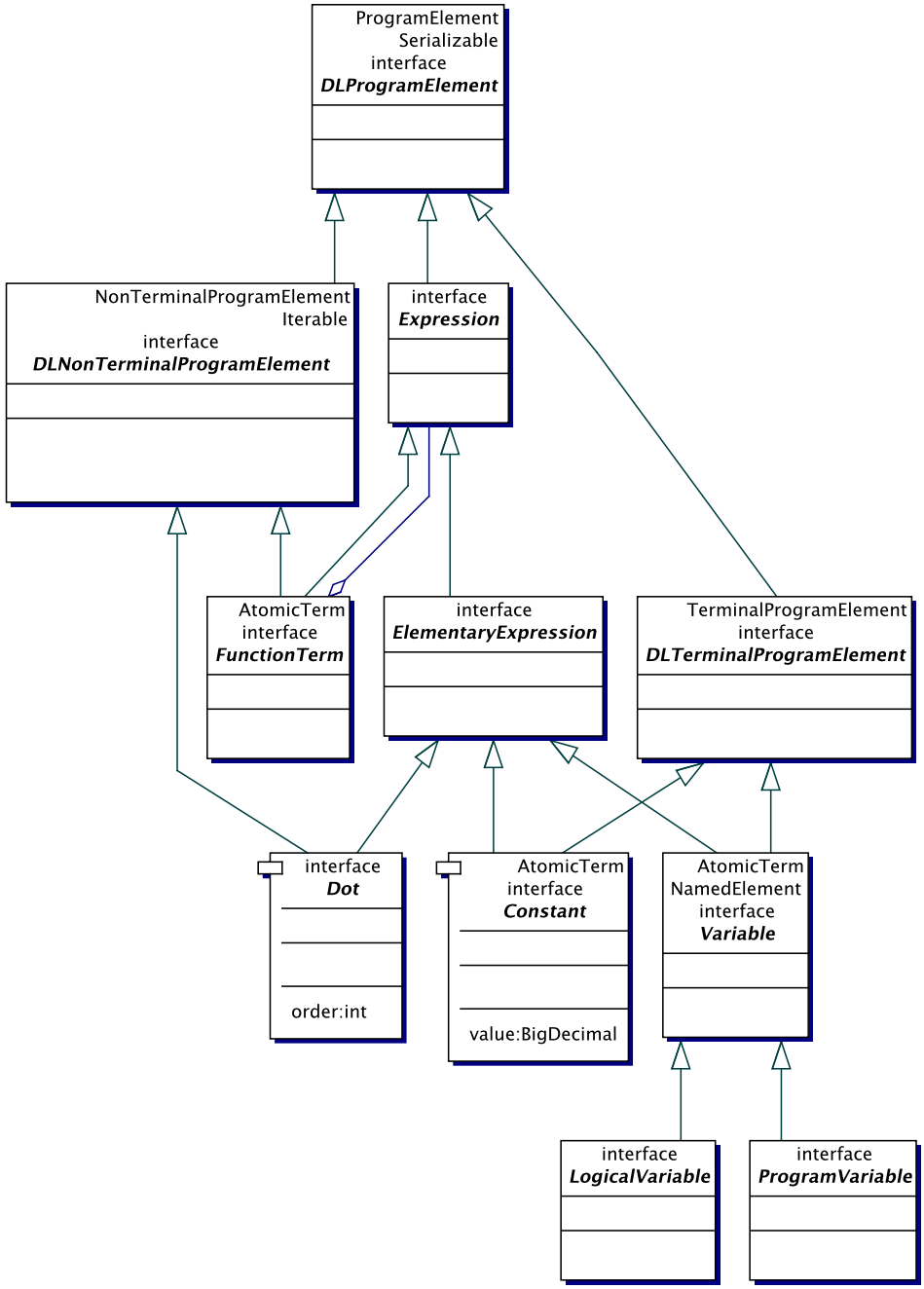


Figure 3.8: Data structures for expressions within dℒ programs generated by α-EXPR, DIFFEQUATION and DIFFEXPR

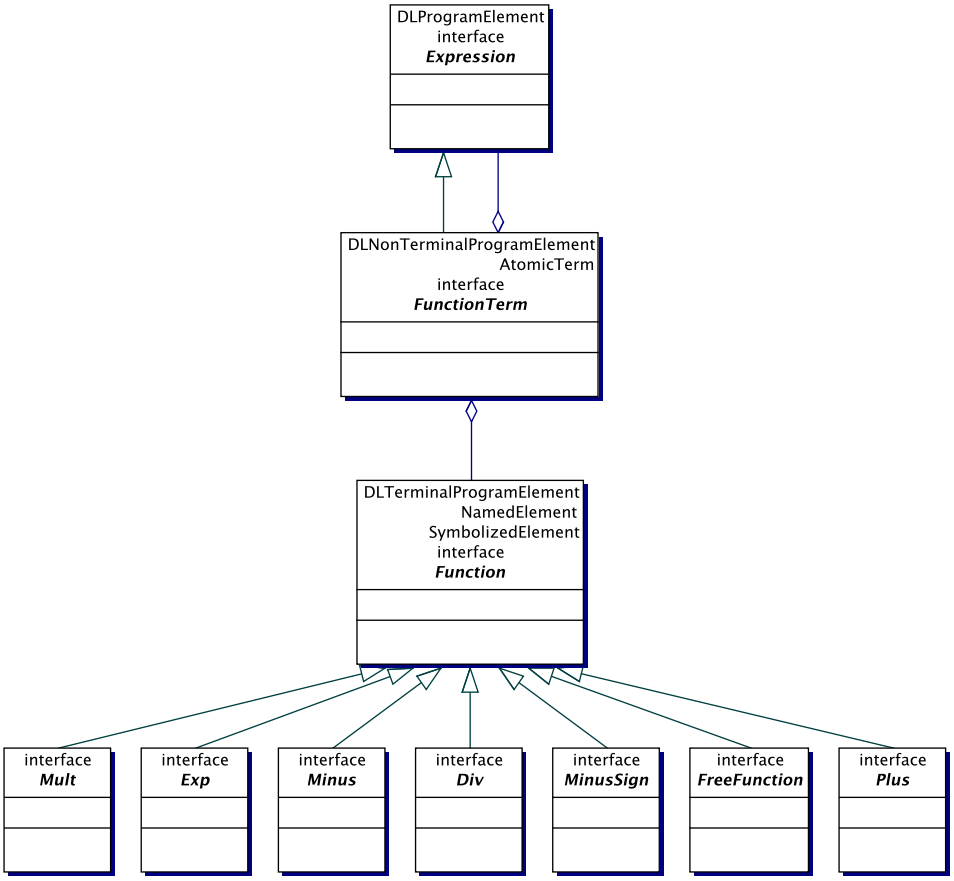
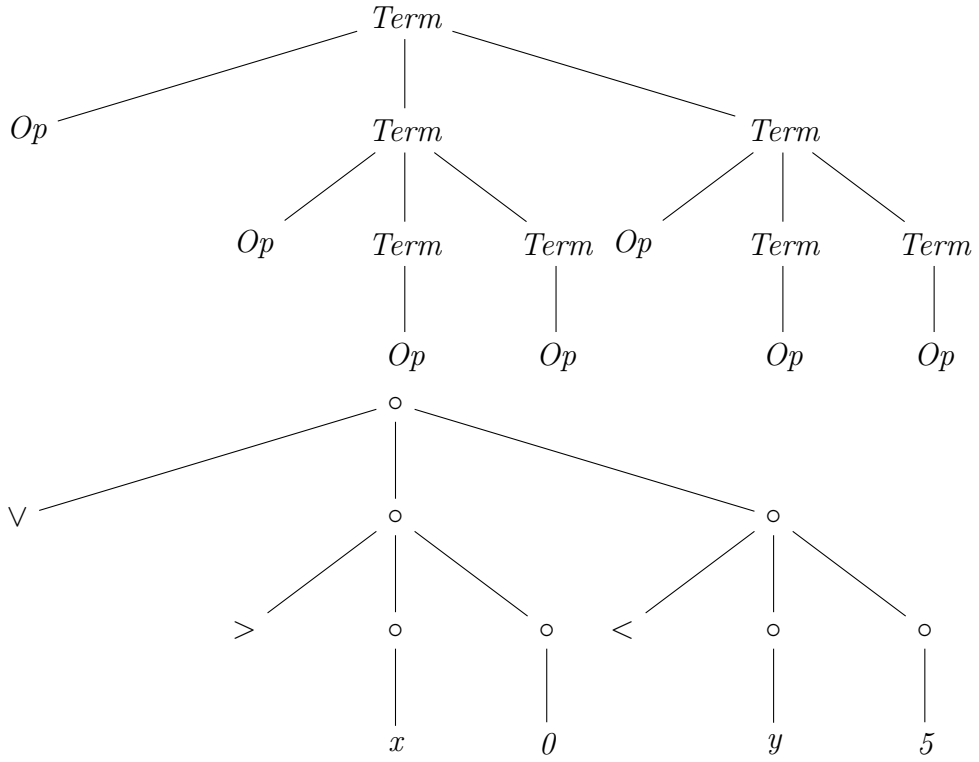


Figure 3.9: Data structures for functions within $\mathrm{d}\mathcal{L}$ programs

the `Term` format. In either case the translation is performed recursively. Using these converters the `MathematicaDLBridge` creates queries in the Mathematica `Expr` format and handles the result. The `MathematicaDLBridge` returns `Term` objects, which are the usual representation of formulas in KeY. The server only has to handle `Expr` objects, thus it is independent from KeY.

The translation is in either case performed by a recursive algorithm that transforms the abstract syntax tree from one representation to another one.

Example 6. *Lets consider the example formula* $x > 0 \vee y < 5$. *In the* ***`Term`*** *representation this is:*



In the ***`Expr`*** *syntax this becomes* `Or[Greater[x,0],Less[y,0]]`.

Another example is the case when Mathematica returns a rational number. In the ***`Expr`*** *syntax this is e.g. represented as* `Rational[1,2]` *for* $\frac{1}{2}$. *We represent this in the* ***`Term`*** *data structures as devision.*

The interface `IMathSolver` provides methods for simplification of formulas, quantifier elimination, solving of differential equation systems and counter example derivation.

Simplification is done using the functions `Simplify` and `FullSimplify` provided by Mathematica. Simplify can optionally take a set of formulas that are assumed to hold. This is also supported by our implementation. The quantifier elimination is done using the function `Reduce`. This function can optionally take a set of variables to eliminate. This is especially useful when handling formulas with a large number of variables. Mathematica is able to solve differential equation systems using the function `DSolve`. We propagate this functionality to handle the continuous evolutions that can occur inside hybrid programs. Another

useful feature is the function `FindInstance`. This function tries to find a satisfying valuation for a given formula. This is useful for finding counter examples, as if there is a satisfying valuation for the negation of a formula, it cannot be valid.

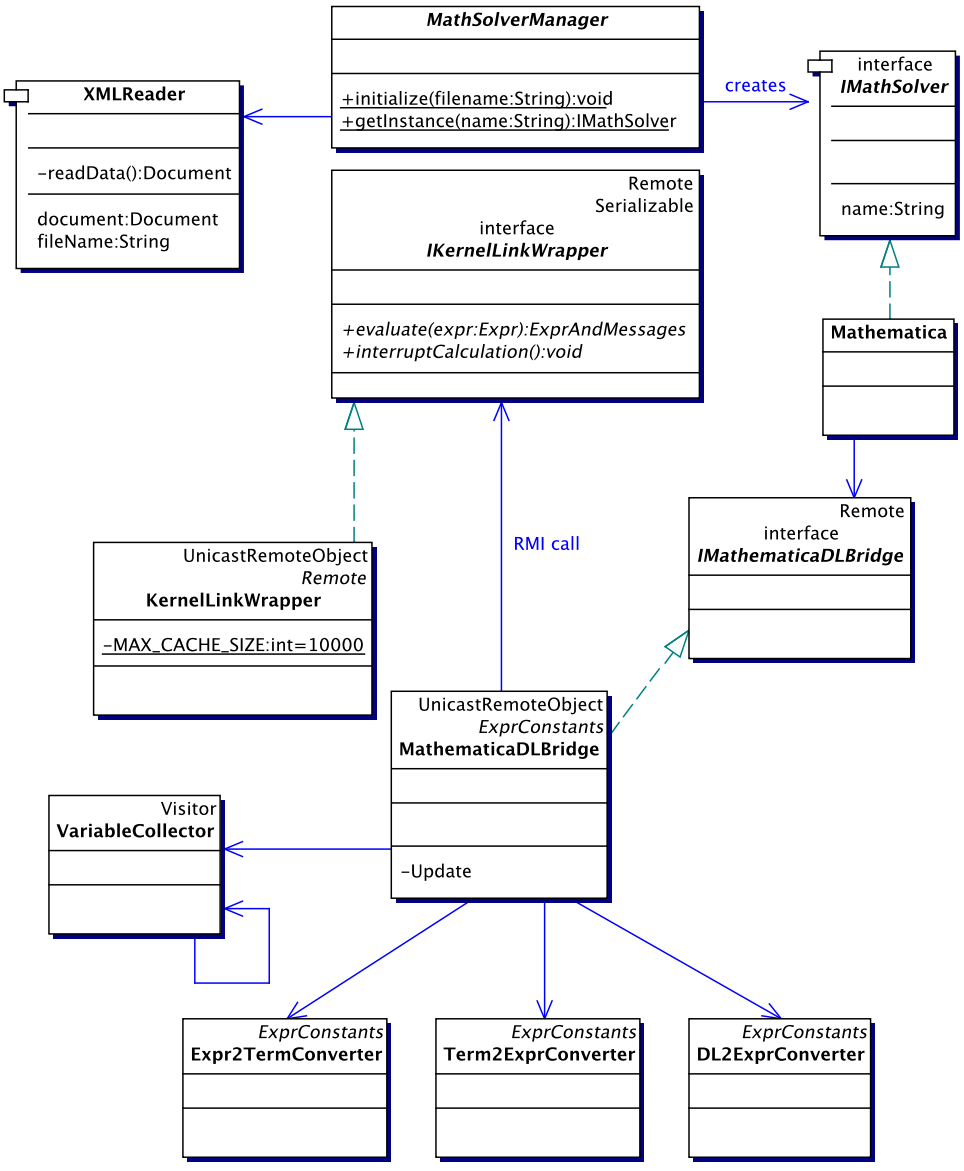


Figure 3.10: Mathematica interface design

Chapter 4

Implementation

4.1 Overview

In this chapter we present implementation details. The chapter structure is the same as in chapter 3; first we will describe the implementation that is related to KeY, followed by a description of Mathematica related implementation details. Additionally, we will describe the input format of KeY and how it can be used in the last section of this chapter.

4.2 KeY Extension

In this section we describe the implementation of the extensions made to KeY. As mentioned earlier, by default the KeY parsers only permit JAVA programs within the modalities. We have extended the parser thus it can handle hybrid programs as well.

4.2.1 Parsing

The problem parser of KeY consists of two parts. The main problem parser can parse the problem file without handling the modalities. For handling the modalities Recoder [GHT07] is invoked. Recoder is a parser for JAVA programs. We decided to add a `ProgramBlockProvider` that is invoked on parsing time and can either handle JAVA, using Recoder, or hybrid programs. Which implementation of the `ProgramBlockProvider` is to be used is determined by the current `Profile`. We created implementations of the `ProgramBlockProvider` for the d$\mathcal{L}$ `Profile` as well as for the JAVA `Profile`. This way our extended version of KeY is still able to check JAVA programs when using the JAVA profile.

For the integration of hybrid programs we have implemented a two stage parser in ANTLR [PQ95]. The first stage produces a classic homogeneous syntax tree. In this regard "homogeneous" donates that there is only one class file for all elements of the tree which are distinguished by a token field (see example 7). As for KeY conventions each element that can occur in a modality has to implement the interface `ProgramElement`, we decided to add a second parsing stage to generate a heterogeneous syntax tree. This stage is realized using the TreeParser mode of ANTLR. The transformation is illustrated in figure 4.1.

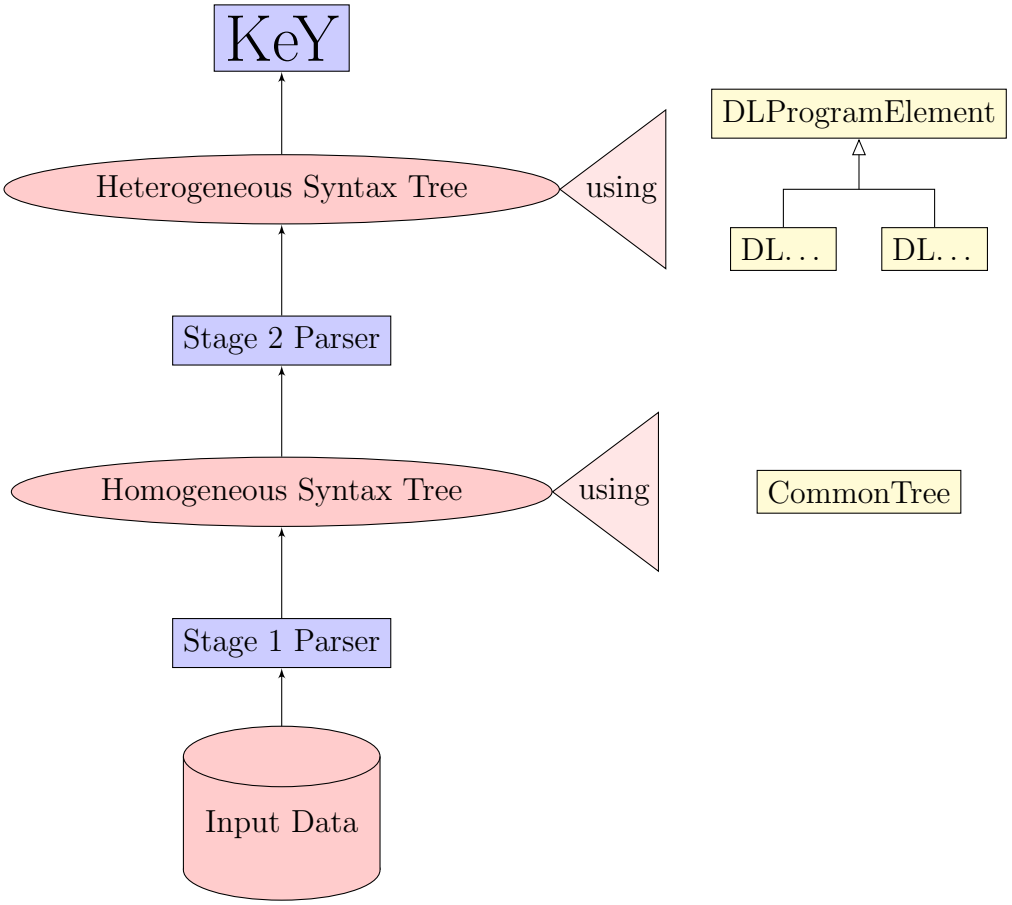
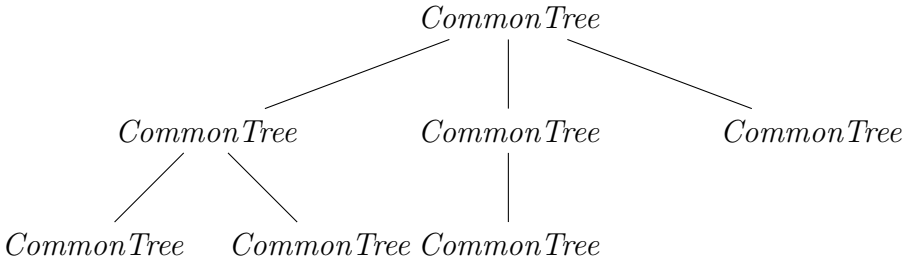


Figure 4.1: Two stage parsing of hybrid programs

Example 7 (Homogeneous Syntax Tree)**.** *The syntax trees generated by ANTLR are homogeneous using only the class* `CommonTree` *this means a syntax tree looks like this:*



Where each of these `CommonTree` *objects holds a* `Token` *for identification.*

The first parsing stage reads the input data passed over by the problem parser. This input data is a string representation of a hybrid program. It is transformed into an homogeneous syntax tree. The second parsing stage is a context free grammar annotated with semantic expressions that produce a representation of the hybrid program using the data structures described in section 3.2.4.

As mentioned in the previous chapter we model the data structures using the abstract factory pattern. We already described the abstract interface structure. In this section we present the concrete implementation of the data structures.

The design of the data structures described in section 3.2.4 is implemented in the classes shown in figures 4.2, 4.3, 4.4, 4.5 and 4.6. Our design contains multiple inheritance. As JAVA does not allow this for classes, the class model is a tree. As the multiple inheritance was manly used for tagging objects, the transformation was simple.

Tha class `DLNonTerminalProgramElementImpl` implements the iterator pattern [GHJV95] (see section A.4.2). This way we can easily access its children using the `foreach` loop introduced in JAVA 1.5. The provided iterator is `DLNonTerminalPrograMElementIteratorImpl`. We have not defined a special interface for it, as the methods provided by the interface `Iterator` are sufficent for our needs. This interface is part of the JAVA class library.

It may be noted that the subclasses of `ComparsionImpl` are all singleton as well as the classes `PlusImpl`, `MinusImpl`, `MinusSignImpl`, `MultImpl`, `DivImpl` and `ExpImpl`. These classes can be implemented using the design-pattern singleton [GHJV95] (see section A.3.3) as they represent mathematical operations whose interpretation or presentation does not change. They are by design used by `PredicateTermImpl` and `FunctionTermImpl` as operators. The classes `FreePredicateImpl` and `FreeFunctionImpl` are used to represent predicates and functions that are not known to the system by default. Functions like *sine* can be integrated as free functions as well. The classes `FreePredicateImpl`, `FreeFunctionImpl` and `ConstantImpl` have integrated caches that assure that there is only one object at a time per name. The class `ConstantImpl` is used to represent numbers. They are represented as `BigDecimal` to keep the representation as accurate as possible.

4.2.2 Calculus Embedding

In this section we describe the integration of an altered version of the calculus described in section 2.4. We decided to keep the same taclets for the propositional part of the formulas and to take on the rules for hiding that already exist for the handling of formulas occurring in a sequent deduction in KeY. This way we could base on the knowledge of the KeY developers.

In table 4.1 we present an overview which class of rules is implemented with which methods. We distinguish tree types of implementations: Rules implemented as taclet, rules implement as taclet using metaoperators and rules implemented as built-in rules. It may be noted that we implemented most of the rules for handling hybrid programs as rewrite rules, as they are local transformations.

We have not implemented an approach for the handling of existential quantifiers on the right side or universal quantifiers on the left side of the sequent yet. The reason for this is that there is no native support for side deductions in KeY and proofs are considered to be trees. Therefore none of the presented approaches for handling these quantifiers could be implemented easily and we decided to focus on a sound implementation of the other rules and a case study that presents the

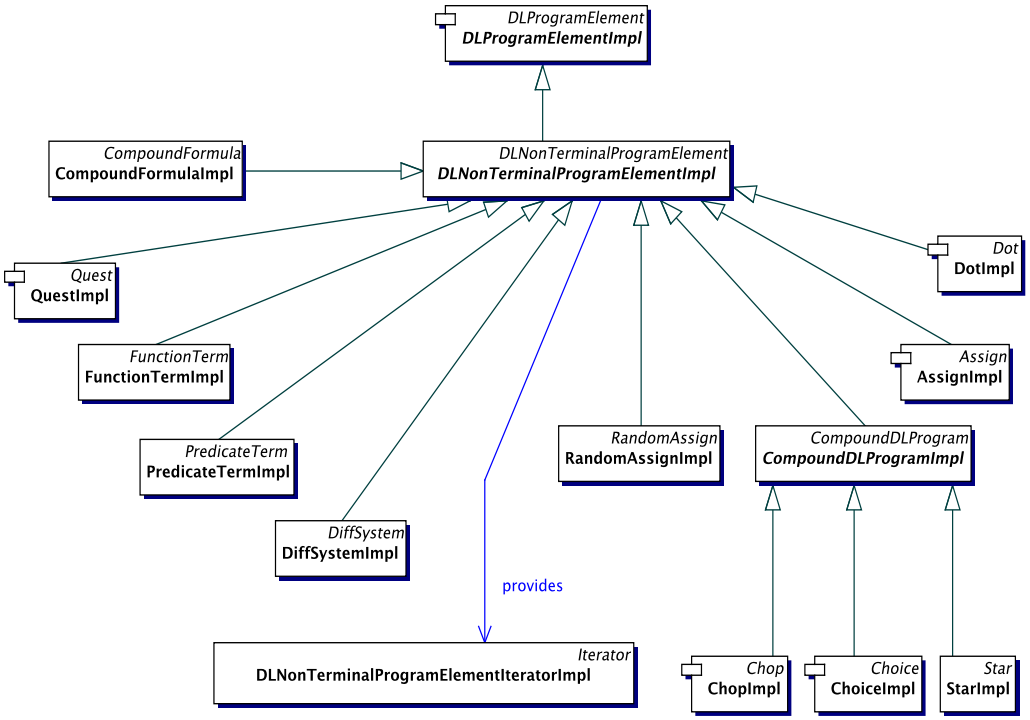


Figure 4.2: Implementation of data structures for non-terminal $d\mathcal{L}$ programs

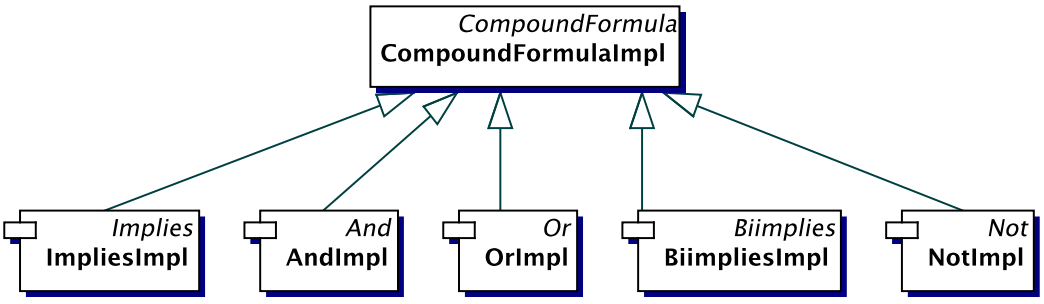


Figure 4.3: Implementation of compound formulas

		Rules for		
		Propositional Logic	FOL	Hybrid Programs
Implemented as	Taclet	x	x	x
	Metaoperator			x
	Built-in		x	x

Table 4.1: Relation between rule classes and implementation method

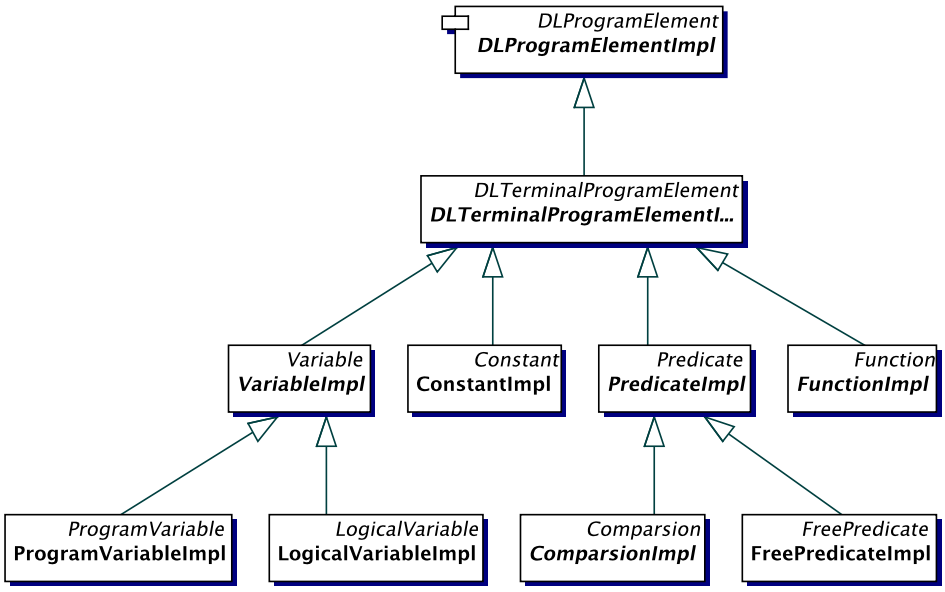


Figure 4.4: Implementation of data structures for terminal $\mathsf{d}\mathcal{L}$ programs

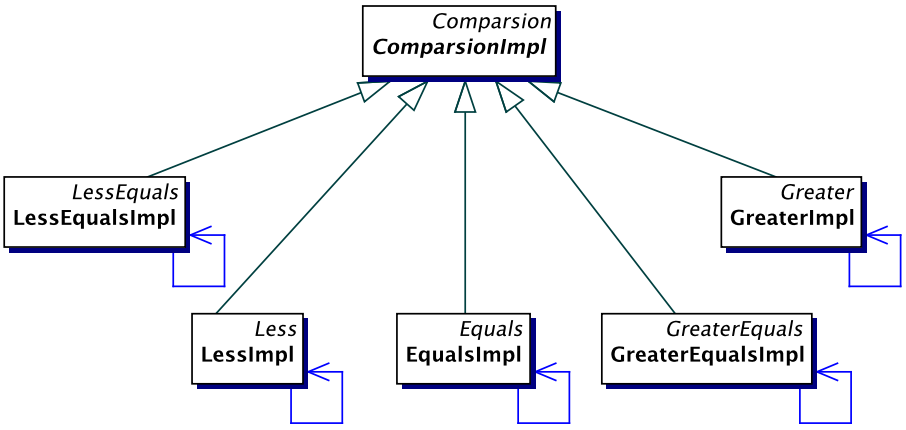


Figure 4.5: Implementation of data structures for comparisons

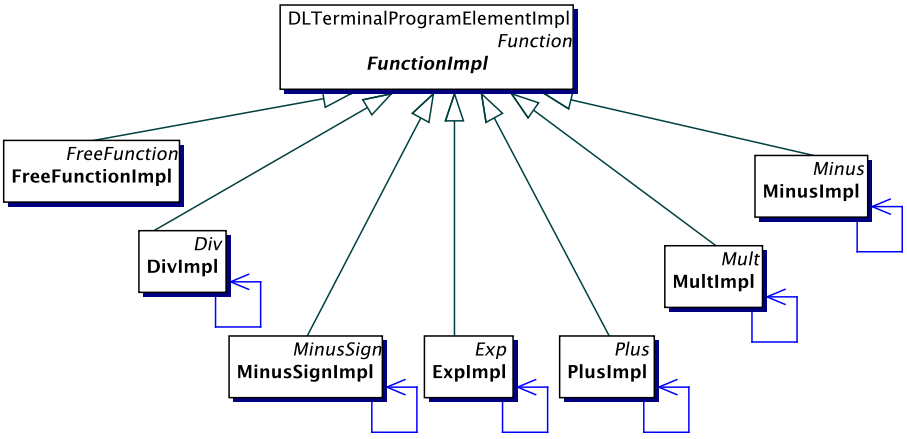


Figure 4.6: Implementation of data structures for functions

expressive power of the implemented fragment. As we will see, some existential quantifiers on right or universal quantifiers on the left side could be handled using other rules already. So a specification that contains these quantifiers is not in general out of the implemented fragment.

The translation of the Skolemization rules R12 and R13 is straight forward. The rule R12 is in taclet representation:

— Taclet —

```
all_right {
        \find (==> \forall u; b)
        \varcond ( \new(sk, \dependingOn(b)) )
        \replacewith (==> {\subst u; sk}b)
        \heuristics (delta)
};
```

— Taclet —

Whereas the rule R13 becomes:

— Taclet —

```
ex_left {
        \find (\exists u; b ==>)
        \varcond ( \new(sk, \dependingOn(b)) )
        \replacewith ({\subst u; sk}b ==>)
        \heuristics (delta)
};
```

— Taclet —

In these taclets 3 schema variables are used: `u` is a schema for a logical variable, `b` is a schema for a formula and `sk` is a schema for a function symbol.

The translation of the rules that handle the sequential composition or the non-deterministic choice can be performed easily as well. They use 5 schema variables: `#allmodal` stands for an arbitrary modality, `#dia` stands for the diamond modality, `#box` for the box modality, `#dl` and `#dl2` are schemata for arbitrary hybrid programs. The rule R19 as taclet looks like this:

— Taclet —

```
modality_split {
        \find (
                \modality{#allmodal}
                #dl;#dl2
                \endmodality(post)
        )
        \replacewith(
                \modality{#allmodal}
                        #dl
                \endmodality(
                        \modality{#allmodal}
                                #dl2
                        \endmodality(post)
                )
        )
    \heuristics(simplify_prog)
};
```

— Taclet —

We represent the non-deterministic choice ∪ with the ASCII [Int83] symbols `++`, thus the rule R20 is translated to the taclet language as follows.

— Taclet —

```
diamond_choice {
        \find (
                \modality{#dia}
                        #dl ++ #dl2
                \endmodality(post)
        )
        \replacewith(
                \modality{#dia}
                        #dl
                \endmodality(post)
                |
                \modality{#dia}
                        #dl2
                \endmodality(post)
        )
   \heuristics(simplify_prog)
```

```
};
```

— Taclet —

For the box case R21 the disjunction is simply replaced by a conjunction.

Some of the elementary programs are handled by simple taclets as well. There are the rules R17 and R18 for handling state assertions, that can be easily translated into the taclets `diamond_quest` and `box_quest`. The schema variable `#dlform` matches a formula inside a hybrid program. It is translated into the corresponding `Term` representation by the `TypeConverter`. This class is responsible for the translation of program elements into logic elements, when they are dragged to the logic level. We extended the `TypeConverter` so that it can convert the formulas occurring in hybrid programs into the common `Term` representation. In the following taclet this schema variable occurs in two contexts. In the `find` clause it represents the formula of a state assertion. In the `replacewith` clause it is on the logical level. As the operators for the conjunction and implication are only defined on formulas (represented by the sort `Sort.FORMULA`), the schema variable has to have a different sort depending on its position. In the program context its sort is `ProgramSVSort` in context of a propositional formula its sort is `Sort.FORMULA`. The schema variable `post` is a schema for an arbitrary $\mathrm{d}\mathcal{L}$ formula.

— Taclet —

```
diamond_quest {
        \find (\modality{#dia}?#dlform\endmodality(post))
        \replacewith(#dlform & post)
        \heuristics(simplify_prog)
};
```

— Taclet —

— Taclet —

```
box_quest {
        \find (\modality{#box}?#dlform\endmodality(post))
        \replacewith(#dlform -> post)
        \heuristics(simplify_prog)
};
```

— Taclet —

For handling the assignment rule R24 we use a taclet as well. It uses schema variables to match both sides of the assignment. On the left side of an assignment there is always a program variable, that is matched by `#dlvar`. The expression on the right side is matched by the schema variable `#dle`.

— Taclet —

```
assignment_to_update {
        \find (
                \modality{#allmodal}
                        #dlvar := #dle
```

```
                \endmodality(post)
        )
        \replacewith(({#dlvar := #dle} post))
        \heuristics(simplify_prog)
};
```

Taclet

This taclet translates the assignment into an update. Updates are then handled by a built-in rule called `UpdateSimplifier`.

4.2.2.1 Metaoperators

Some rules cannot be expressed directly as taclets but they have certain locality properties such that we can use a metaoperator to get them into a taclet. A metaoperator passes its name to its superclass which is invoked by the parser to find out which expressions can be handled as a metaoperator. A metaoperator further has a `caclulate` method that is invoked on taclet application. Its parameters are the currently matched term, the instantiations of the schema variables in the taclet, as well as the `Services` for configuration reasons.

One example for this is the `ODESolve` rule. It is an extended implementation of the rules R27 and R28. As it only changes the program it can be written down using a metaoperator. This metaoperator performs the whole transformation. It adds a new variable T to be quantified and calls `DSolve` in Mathematica. The extension made by this rule is, that it can handle systems of differential equations and invariants, instead of a single differential equation. This extension is sound [Pla07c].

Taclet

```
ODESolve {
        \find (
                \modality{#allmodal}
                        #diffsystem
                \endmodality(post)
        )
        \replacewith(
                #ODESolve(
                        \modality{#allmodal}
                                #diffsystem
                        \endmodality(post)
                )
        )
        \heuristics(simplify_prog)
};
```

Taclet

The rules for calls of `Simplify`, `FullSimplify` and `Reduce` also use metaoperators. The random assignments are handled by a metaoperator as well, as it is

not possible to insert new quantified variables in taclets and KeY does not allow quantifiers over program variables. The same holds for unwinding of loops.

The metaoperators we added are shown in figure 4.7 and we will describe in short what they are supposed to do.

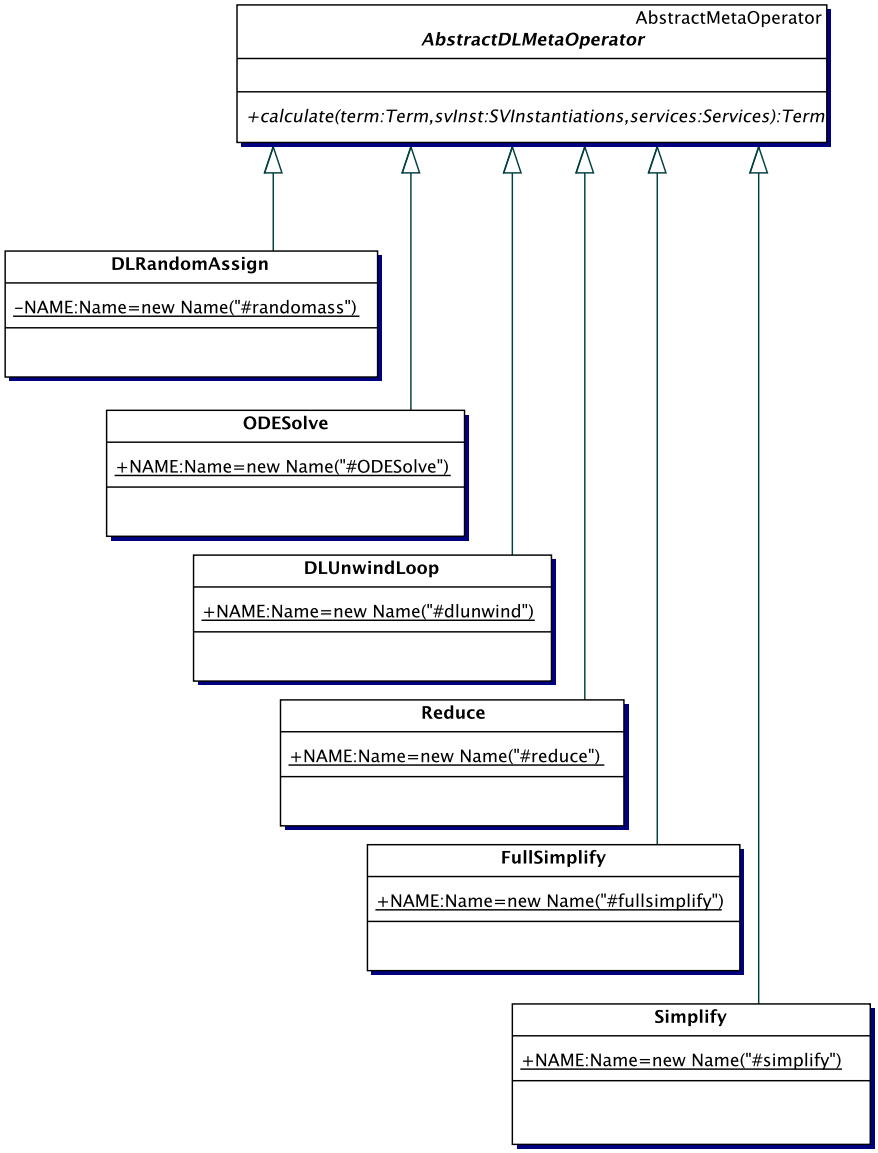


Figure 4.7: Metaoperators used in the taclets for $\mathrm{d}\mathcal{L}$

`#dlunwind` is used to unwind a loop. It is needed for the implementation of the rules R22 and R23.

`#ODESolve` is used to solve differential equation systems. It calls the `DSolve` function of the arithmetic solver and transforms the result into an update.

`#randomass` is the metaoperator that handles random assignments. It introduces a new quantifier depending on the modality type. An update is introduced that replaces the randomly assigned variable with the newly introduced quantified one.

`#simplify` wraps the `Simplify` function of Mathematica that performs simplification depending on a given cost function on the given term. It can handle a complete formula as well as an expression.

`#fullsimplify` wraps the `FullSimplify` function of Mathematica. `FullSimplify` works basically the same as `Simplify` but it performs more expensive transformations.

`#reduce` is the most important metaoperator. It is used as interface to the quantifier elimination which is implemented in the Mathematica function `Reduce`. The metaoperator is as the simplification operators, applicable on either a complete formula or on a part of it.

The implementation of rule R29 (see figure 4.8) uses a metaoperator (`#introNewAnonUpdate`) for the introduction of an anonymous update, that renames all variables that are changed by the loop. This way we can keep all formulas in the sequent, thus the invariant does not have to include propositions about constant values, as they are already included in the current formulas.

— Taclet —

```
loop_inv_box {
  \find (==> \[#dl*\]post)
  \varcond(\new(#modifies, \dependingOnMod(anon1)),
           \new(#modifies, \dependingOnMod(anon2)))
  "Invariant␣Initially␣Valid":
    \replacewith (==> inv );
  "Body␣Preserves␣Invariant":
    \replacewith (==>  #introNewAnonUpdate(
                #modifies, (inv -> \[#dl\]inv), anon1));
  "Use␣Case":
    \replacewith (==> #introNewAnonUpdate(
                #modifies, inv -> post, anon2))
    \heuristics (loop_invariant, loop_invariant_proposal)
    \displayname "loop_invariant"
};
```

— Taclet —

Figure 4.8: Taclet representation of the invariant rule R29

The rule implemented here is an induction rule. Loops can be handled by finding an invariant that is strong enough to imply the post condition. The invariant has to be initially valid, because repeating the loop zero times is valid for the non-deterministic repetition. Also we have to proof that the invariant is preserved by the body of the loop. This is the induction step. At last, we have to show that the invariant is strong enough to imply the post conditions.

For the induction step and the test whether the invariant is strong enough the previous context would have to be dropped. As this would not be possible using the taclet language, the trick is to rename the variables that are changed by the loop body in these cases. They are renamed by the metaoperator `#introNewAnonUpdate`.

After applying this rule it is useful to apply hiding rules to the formulas that cannot lead to closure of the induction step branch. Usually there are formulas expressing constraints for non-rigid variables that are initially true. As the induction step talks about another state, these variables are renamed, thus these formulas do not contain useful informations anymore. As Mathematica does not know that these formulas are only left-overs, a `reduce` call tries to satisfy them as well, which is not necessary at all.

Example 8. *Let us consider a simple example. If we want to proof that*

$$((x \geq b \land b < 0) \rightarrow [(x := x + 1)^*]x \geq b)$$

holds, we have to proof three propositions.

1. $((x \geq b \land b < 0) \rightarrow x \geq b)$
2. $((x \geq b \land b < 0) \rightarrow \mathcal{R}(inv \rightarrow [x := x + 1]inv))$
3. $((x \geq b \land b < 0) \rightarrow \mathcal{R}(inv \rightarrow x \geq b)$

Here $\mathcal{R}$ is the anonymous update that renames all occurrences of the variables changed by the loop. This is in this case only x.

The invariant is changed by the anonymous update as well as it talks about the new state. The formula $x \geq b$ on the left side of the implication only states a proposition about the initial state. The loop invariant has to provide the necessary informations about other states. On the other hand the variable b is not changed in the loop body, therefore the constraint $b < 5$ is preserved and b is not renamed by the update.

Proof. With the obvious invariant for this system $\text{inv} :\Leftrightarrow x \geq b$ the proof can easily be performed. The first case states an obvious tautology.

$$\dfrac{\dfrac{\dfrac{}{x \geq b, b < 0 \ \vdash x \geq b}\,{}^{*}}{x \geq b \land b < 0 \ \vdash x \geq b}\,\text{R11}}{\vdash ((x \geq b \land b < 0) \rightarrow x \geq b)}\,\text{R6}\,\text{R3}$$

Assuming the anonymous update renames the variable x to x_2, the second case becomes:

$$((x \geq b \land b < 0) \rightarrow (x_2 \geq b \rightarrow [x_2 := x_2 + 1]x_2 \geq b))$$

Which can be proven valid with:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x \geq b, b < 0, x_2 \geq b \vdash x_2 + 1 \geq b}\,*}{x \geq b, b < 0, x_2 \geq b \vdash [x_2 := x_2 + 1]x_2 \geq b}\,\text{R24}}{x \geq b, b < 0 \vdash (x_2 \geq b \rightarrow [x_2 := x_2 + 1]x_2 \geq b)}\,\text{R3}}{x \geq b \land b < 0 \vdash (x_2 \geq b \rightarrow [x_2 := x_2 + 1]x_2 \geq b)}\,\text{R6}}{\vdash ((x \geq b \land b < 0) \rightarrow (x_2 \geq b \rightarrow [x_2 := x_2 + 1]x_2 \geq b))}\,\text{R3}$$

The third goal is after applying the anonymous update:

$$((x \geq b \land b < 0) \rightarrow (x_2 \geq b \rightarrow x_2 \geq b)$$

Which can be proven valid as well:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x \geq b, b < 0, x_2 \geq b \vdash x_2 \geq b}\,*}{x \geq b, b < 0 \vdash x_2 \geq b \rightarrow x_2 \geq b}\,\text{R11}}{x \geq b \land b < 0 \vdash x_2 \geq b \rightarrow x_2 \geq b}\,\text{R3}}{\vdash (x \geq b \land b < 0) \rightarrow (x_2 \geq b \rightarrow x_2 \geq b)}\,\text{R6}$$

□

This example also shows that the proof targets often become pure arithmetic at the end. These targets are handled using the quantifier elimination.

4.2.2.2 Built-in Rules

A central built-in rule of the KeY prover is the `UpdateSimplifier`. The `UpdateSimplifier` is used by a built-in rule to process updates. It applies the updates if this is possible or merges two updates into parallel updates [BHS07]. Therefore it can be considered the implementation of our calculus rule R24. By default the `UpdateSimplifier` cannot apply an update to a modality, thus it applies the update to the part of the formula before the modality and moves the update in front of the modality.

This design decision was made by the KeY developers as applying updates to JAVA programs is difficult, as expression may for example contain the increment operator `++`. Also it would be necessary to evaluate the influence of all method calls on the variable that should be update.

As it is much easier in the context of hybrid programs, to determine which variables are altered, we extended the `UpdateSimplifier` according to the definition of the application of assignments (definition 32 on page 21). This extension is implemented in the class `DLApplyOnModality`. Our extension additionally covers the case that a parallel update should be applied to the modality.

The built-in rules, we implemented, are shown in figure 4.9.

A rule has been created that uses the decision procedures of the arithmetic solver. It is called `ReduceRule` or `ReduceSequence`. The rule works as follows. First, it reassembles the implication expressed by the sequent, i.e. for a sequent $\Gamma \vdash \Delta$ the formula

$$(\bigwedge_{\varphi \in \Gamma} \varphi) \rightarrow (\bigvee_{\psi \in \Delta} \psi)$$

is created. Afterwards the `reduce` function of the arithmetic solver on this implication. The `reduce` function reduces a given formula by solving equations or inequalities and eliminating quantifiers.

Only formulas that are already first-order are used for the `reduce` call. Formulas that contain substitutions, modalities or updates are ignored, i.e. left unchanged and added to the resulting sequent. The result calculated by Mathematica is translated into the `Term` representation and added to the right side of a new sequent. If the result is `true` the goal is closed directly by the `ReduceRule` rule. For a proof that it is sound to pick up some formulas process them and leave the context unchanged see lemma 3. This lemma can be applied as `reduce` yields an equivalence transformation of the input formula. In fact we assume that it yields an equivalence as its a black-box operation imported from a commercial tool. We want to point out at this point that this rule is an integral part of our implementation and the usage of `reduce` endangers completeness as well as soundness. Lets consider a simple example.

Example 9. *If* `reduce` *would produce for the input formula* $x > 0$ *the output formula* $x < 0$ *the implementation of the* `ReduceRule` *would not be sound.*

If `reduce` *would produce always produce the output* $false$*. The implementation would be sound, but the application would yield a crucial loss of information.*

It has to be noted that there is no corresponding calculus rule to the `ReduceRule`.

The `EliminateQuantifierRule` is the implementation of rule R14. It can be applied by clicking on the Skolem symbol that should be eliminated.

Both of these rules work like the `ReduceRule` but the implication is only constructed for formulas that contain the Skolem symbol to eliminate. The rule `EliminateQuantifierRuleWithContext` works like the `EliminateQuantifierRule` but uses the contextual formulas as well. Both rules reintroduce the quantifiers that existed before the Skolemization. The behavior of the `EliminateQuantifierRule` could be simulated by the `EliminateQuantifierRuleWithContext` with using hiding rules on the unrelated formulas in the sequent by hand. As this is not comfortable and in some cases the quantifier can be eliminated easily without considering the contextual formulas, we decided to keep both implementations.

The `reduce` call may contain a set of variables to eliminate. The user is asked to enter this set. If `reduce` is called with an empty set, Mathematica tries to determine a reasonable set of variables itself.

In the automatic mode the set of variables that should be eliminated is empty for the application of the `ReduceRule`. For the `EliminateQuantifierRule` the

singleton set contains the symbol that is to be eliminated. This set is also used as suggestion when applying the rule manually. The `EliminateQuantifierRuleWithContext` is not used by our strategy.

The `FindInstanceRule` is no real calculus rule. It is used to produce counter examples for a given sequent. It calls `FindInstance` on the negation of the current sequent and shows a popup with the result of the arithmetic solver. `FindInstance` tries to find a valuation that satisfies the given formula. If it returns with a result for the negation of the current goal, we got a counter example. This rule can be used to test e.g. if a formula can be hidden, if an invariant is sufficient or to get an idea of which system behavior can occur that is undesired.

4.2.3 Integration Challenges

PrettyPrinter As we changed the representation of the programs, it was necessary to change the components that are reliable for the presentation of the formulas to the user. The visualization of the formulas, at least the program part, is done by a visitor [GHJV95] called `PrettyPrinter`. The `PrettyPrinter` constructs a human readable representation of the program. It constructs a linear string that contains, e.g. parenthesis which are not necessary in the data structures. Additionally, it defines regions that can be highlighted for user interaction. This way subformulas, or in this case subprograms are identified. The implementation available is specifically designed to print out JAVA programs. We have added some methods so the hybrid programs can be formatted too, but it would be a much better design to separate the visitors. This could be transformed in a design concept to create interfaces for several subclasses of the visitor and alter the `prettyPrint` methods according to the principle "If the visitor can handle me, e.g. is instance of `IfPrettyPrinter`, use the special method, otherwise perform the default action". But this would require a major change of design change which would be difficult to realize due to the fact that the `PrettyPrinter` is not instantiated at a central location.

Substitutions The substitutions provided by KeY only affect the FOL part of the formula. As in $\mathsf{d}\mathcal{L}$ it is possible to have logic formulas in programs, we had to extend the substitution in order to assure that it can also be applied to programs. We extended the substitutions such that they match the definition provided in lemma 1 on page 11.

This is especially required for the Skolemization rules (R12 and R13) as they use the substitution to introduce the Skolem symbol. The extension of the substitution is implemented as a translation of the program where inner nodes of the syntax tree are cloned and unchanged leaves are kept as the same objects. This is done to preserve fast matching of programs and to save memory.

4.2.4 Strategy

For automation of proofs KeY provides so called proof strategies [BHS07]. A strategy in KeY assigns a cost value to each applicable rule. The *cheapest* appli-

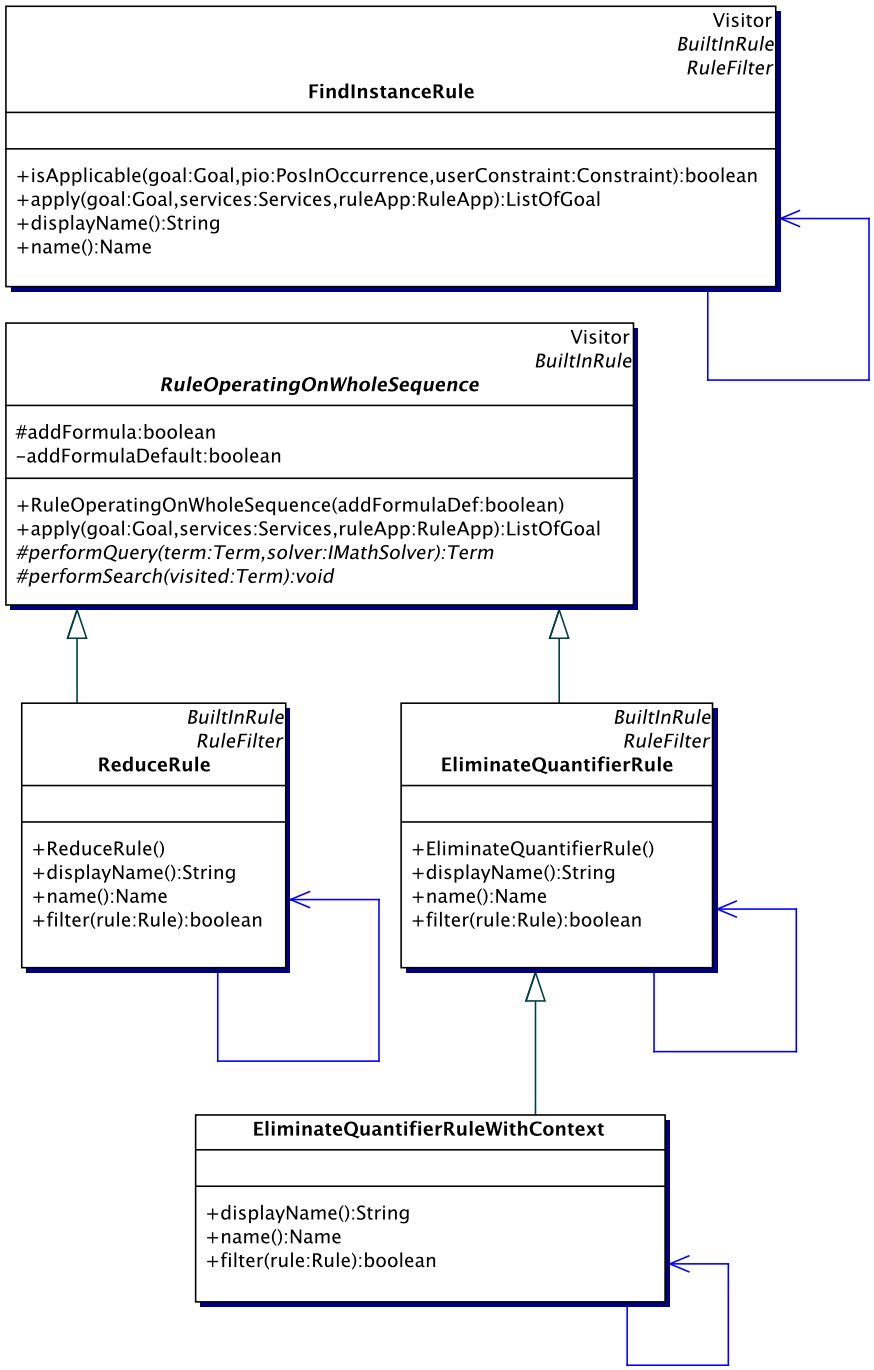


Figure 4.9: The built in rules used for the $\mathrm{d}\mathcal{L}$ calculus

cable rule will be applied.

A naive approach to implement this would be using one priority queue per goal and fill these with all applicable rules that are applicable on the specific goal. The cheapest rule gets the highest priority. A goal is randomly chosen and the rule with the highest priority on that goal is applied and afterwards removed from the queue. Now the rules are added to the queue which have become applicable due to changes made by the applied rule. Additionally, those rules are removed which are not applicable anymore.

We must admit that the implementation is in fact more complicated but this should naive approach should illustrate its basic idea.

Definition 38 (Strategy)**.** *Let* Υ *be the set of all sequents of the form* $\Gamma, \Gamma' \vdash \Delta, \Delta'$*,* $\Re$ *the set of all rules of the form*

$$\{(\Gamma_1 \vdash \Delta_1), \ldots, (\Gamma_n \vdash \Delta_n), (\Gamma' \vdash \Delta')\}$$

and Π *the set of all unfinished proofs. A* strategy *is a function that assigns a cost to each rule application to a sequent in an unfinished proof*

$$Strat : (\Upsilon \times \Re \times \Pi) \mapsto \mathbb{R}$$

We have extended the FOL proof strategy that comes with KeY. Our strategy assigns low costs to rules for propositional logic as well as for the Skolemization rules. This is done, as the application time of these rules is short and in most cases they produce sequents that can be handled easier by the other rules.

The built-in rules we added all implement the interface `RuleFilter`. This is done to be able to use them within a strategy. The interface `RuleFilter` is used to identify classes of rules. There is a default feature that evaluates the `RuleFilter` and assigns costs depending on the result.

If the propositional structure of the formula already yields a satisfying solution for closing a branch it is not necessary to expand the program elements. Therefore the costs for handling program elements are a bit higher and rules for eliminating quantifiers or reduction of the whole sequent are very expensive.

This way the time expensive operations like the `reduce` call to Mathematica are only applied to rather simple formulas. The user can use options to specify whether sequents containing only FOL formulas should be directly passed to the arithmetic solver or if they should be processed further by KeY and `ReduceSequence` is used as last resort. We provide these two possibilities as in general it should be possible for Mathematica to close a goal at the moment when only first-order formulas are left, but as the sequent might be still too complicated the user can choose to process it further using other calculus rules. The goals resulting from revealing the propositional structure yield in many cases sequents that can be processed faster by the reduction algorithm. By default the branches are not expanded after every formula in the branch is a first-order formula, as otherwise the proof may branch many times unnecessarily. While performing the case study we have remarked that some goals can be closed by Mathematica directly within a few seconds or split into several thousand sub goals before closing each separately.

Example 10. *The sequent* $\Gamma, x > b, b > 0 \vdash x > 0, \Delta$ *is obviously valid, but depending on the context* Γ *and* Δ *the proof might split several times if we do not apply the* `ReduceRule` *to this sequent.*

We allow the strategy to call `reduce` on a FOL formula if its toplevel operator is a quantifier. This way we can handle universal quantifiers on the left side of the sequent and accordingly existential quantifiers on the right side if the formula is already first-order. This is necessary as we have not implemented a general approach for the elimination of these quantifiers yet. Afterwards `simplify` on the same formula should be called as `reduce` does in most cases return formulas that result in a lot of branches. Calling simplify solves this problem, as it removes unnecessary branching. The `simplify` rule should also be applied with a very high priority, i.e. very low costs, to formulas containing only numbers as ground terms. This way we can omit the splitting of sequent that contains e.g. $3 = 3$ on the right side. We have realized this by creating a new `Feature` that assigns low costs to the simplify rule, if `reduce` was applied but simplify is not applied yet or if the formula does only contain numbers as ground terms.

While performing the case study, we have observed that some kind of normalization of the inequalities could be useful. Therefore we have added rules that normalize the inequalities such that they are all on the left side of the sequent and only contain $>$ or $\geq$. For this we added the rules presented in figure 4.10. The soundess of these rules is a direct consequence from the definition of the sequent symbol (definition 28) and the semantics of the inequality symbols. This way we could easily provide rules for closing branches if complementary premisses occur.

$$\text{(R30)}\quad \frac{\Gamma, \theta_1 \geq \theta_2 \vdash \Delta}{\Gamma \vdash \Delta, \theta_1 < \theta_2} \qquad \text{(R32)}\quad \frac{\Gamma, \theta_2 > \theta_1 \vdash \Delta}{\Gamma, \theta_1 < \theta_2 \vdash \Delta} \qquad \text{(R34)}\quad \frac{\Gamma, \theta_2 \geq \theta_1 \vdash \Delta}{\Gamma \vdash \Delta, \theta_1 > \theta_2}$$

$$\text{(R31)}\quad \frac{\Gamma, \theta_1 > \theta_2 \vdash \Delta}{\Gamma \vdash \Delta, \theta_1 \leq \theta_2} \qquad \text{(R33)}\quad \frac{\Gamma, \theta_2 \geq \theta_1 \vdash \Delta}{\Gamma, \theta_1 \leq \theta_2 \vdash \Delta} \qquad \text{(R35)}\quad \frac{\Gamma, \theta_2 > \theta_1 \vdash \Delta}{\Gamma \vdash \Delta, \theta_1 \geq \theta_2}$$

$$\text{(R36)}\quad \frac{}{\Gamma, \theta_1 > \theta_2, \theta_2 > \theta_1 \vdash \Delta}$$

$$\text{(R37)}\quad \frac{}{\Gamma, \theta_1 > \theta_2, \theta_1 = \theta_2 \vdash \Delta}$$

$$\text{(R38)}\quad \frac{}{\Gamma, \theta_1 > \theta_2, \theta_2 = \theta_1 \vdash \Delta}$$

- θ_1, θ_2 are terms

Figure 4.10: Rules for normalization of inequalities

4.3 Mathematica Integration

In this section we describe implementation details of the integration with Mathematica.

Exception Handling As mentioned in chapter 3 we use a client server architecture. The server functions are accessed using RMI. The RMI implementation demands that every function that is used remotely may throw a so called `RemoteException`. These exceptions are transported from the server to the client automatically. If errors occur during conversion or during the calculation step a `RemoteException` is thrown. In most cases the `RemoteException` is used as container exception. The exceptions thrown by J/Link, are packed into the `RemoteException` and passed to the client. The implementation of our built-in rules performs all operations that need Mathematica before altering the proof, thus when an exception occurs during processing of a built-in rule, it does not alter the proof. Unfortunately this does not work for the rules using Metaoperators. In case of an exception the result of the Metaoperator is identical to the input, but if the user does not undo this proof step, the resulting proof is inconsistent, as the rule was not really applied. If Mathematica returns `$Aborted` an exception is thrown manually as well. The J/Link documentation says that the interface should raise an exception by itself but runtime analytics have shown that no exception occurs and an `Expr` object is returned containing the `String` `$Aborted`.

Caching For performance reasons the server caches the queries. Up to 10000 queries are cached in a `HashMap` that maps queries to results. Before invoking Mathematica, the cache is checked whether the result is already known.

This is done because analysis have shown that there are many similar queries. This happens frequently if there is a quantified formula somewhere in the sequent and another formula causes the proof to split the sequent into two branches. The quantifier is now eliminated at least once per branch probably resulting in exactly the same queries. The Skolemization rules R12 and R13 could brake this symmetry in queries but in many cases the cache results in a huge performance gain. This results from the fact that KeY generates names that are unique on the whole proof. Therefore the introduced Skolem symbol does have another name, thus the query is performed again.

Conversion Challenges In order to translate formulas to `Expr` it is essential to convert the numbers into the right format. If there is a real number in the `Expr` Mathematica uses numerical methods for the evaluation, whereas symbolic methods are used for rationals. Mathematica infers a result accuracy from the input values. Therefore if numbers are passed to Mathematica in floating point representation, Mathematica uses numerical methods with a certain accuracy. As we want to apply symbolic methods whenever possible we use objects of the type `BigDecimal` to represent our numbers, as they make it easy to decide whether

a number is a whole number or not. If Mathematica returns rationals, we store them as fraction using the division operator. This way we keep the information instead of converting it into a floating point number.

4.4 Usage

In this section we describe how the tool can be used. First we describe the input format and afterwards we provide an overview over the control elements in the user interface.

4.4.1 Input Format

The input files for KeY are plain text files using a certain structure. For $\mathsf{d}\mathcal{L}$ there is the declaration of sorts at the beginning of the file. This section should only contain R yet. This sort is used to represent the real numbers $\mathbb{R}$. Afterwards there may be the declaration of the function symbols. The last section defines the proof obligation called `problem`.

The following listing shows a EBNF grammar of an input file format as it is used for $\mathsf{d}\mathcal{L}$.

— KeY Syntax —

```
problemfile     ::=
                sorts
                functions?
                problem

sorts           ::= "\sorts␣{␣R;␣}"

functions       ::= "\functions" "{" (functiondecl ";")* "}"

functiondecl    ::= "R" name (paramlist?)

paramlist       ::= "(R" (",␣R")* ")"

problem         ::= "\problem" "{" formula "}"
```

— KeY Syntax —

Example 11 (Problem File)**.** *The following listing contains an example problem file. The statement* `\[ x:= x\]` *is used as a declaration of the variable* `x`*. It is necessary if a program variable is used in the first-order part before occurring in a modality. In this example it could be omitted, as the variable is only used in the FOL part in the scope of the diamond modality containing the assignment. It is only added here for exemplification.*

— *KeY Problem File* —

```
\sorts {
```

```
        R;
}

\functions {
        R b;
        R f(R,R);
}

\problem {
        \[x:=x\] (b > 0 -> \<x:=f(b,b)\> x = f(b, 1))
}
```

KeY Problem File

The input syntax for formulas is slightly different from the abstract syntax presented in section 3.2.2. One reason is that it has to handle parenthesis, but there is also the problem, that e.g. the symbol $\cup$ is not available on a standard PC keyboard. Also the Newton Notation for the derivatives is hard to enter, so we decided to use other symbols to solve this issue.

The part that can already be parsed by the default KeY parser uses some kind of LaTeX notation for the quantifiers. The universal quantifier is written as `\forall` and the existential quantifier is written as `\exists`. The modalities are written as `\[ alpha \]` for the box modality and `\< alpha \>` for the diamond modality.

We decided to use `++` to donate the non-deterministic choice. The notation of the derivatives is similar to the Lagrange Notation but we keep the implicit knowledge that it is always the derivative by respect to the time. Instead of $\dot{x}$ we write `x‘`. It has to be noted that this symbol is an accent not a single quote, as single quotes are handled specially by the KeY lexer, thus we cannot use them.

We should also remark that KeY demands that variables are declared before they are used. A program variable is considered to be declared if its in the scope of a modality that refers to the variable. If it is necessary to define preconditions for the variables one could use the trick to define the variable by adding a modality that states `\[ x:= x\]`. This modality does not effect the value of the variable but under the scope of this modality the name of the variable is defined and can be used. The mapping between abstract syntax and input syntax is illustrated in table 4.2. It has to be noted that for marking derivatives an accent is used. We cannot use a single quote here as the KeY lexer uses single quotes to identify character literals.

4.4.2 Tool Overview

In figure 4.11 a snapshot of the KeY prover is shown. The GUI is programed in Java Swing [Sun05, HWL+02], thus it has a traditional look and feel. In the left upper corner, we see the open problem files in the box titled `Tasks`. Below the `Tasks` box there are some tabs that are related to the current proof. The tab

Abstract	Input String
$\forall$	`\forall`
$\exists$	`\exists`
$\wedge$	`&`
$\vee$	`|`
$\rightarrow$	`->`
$\leftrightarrow$	`<->`
$[]$	`\[\]`
$<>$	`\<\>`
$?$	`?`
$:=$	`:=`
$<$	`<`
$\leq$	`<=`
$=$	`=`
$\geq$	`>=`
$>$	`>`
$+$	`+`
$-$	`-`
$\times$	`*`
$\div$	`/`
x^y	`x^y`
$\dot{x}$	`x‘`
{DIFFSYSTEM}	`{diffsystem}`

Table 4.2: Mapping from abstract syntax to input syntax

with the label `Proof` shows the proof tree. Goals marked green are closed ones. The tab `Goals` is a list view of the current goals. The `User Constraints` are not used in the dℒ implementation. Normally they are invoked when closing a branch to choose a substitution matching certain conditions. The `Proof Search Strategy` tab is used to choose the strategy that should be applied when pressing the *Play* button. In the dℒ case only the dℒ strategy is available at the moment. The last tab provides a overview over the available calculus rules. On the right side we see the current sequent and some meta informations like which rule is to be applied.

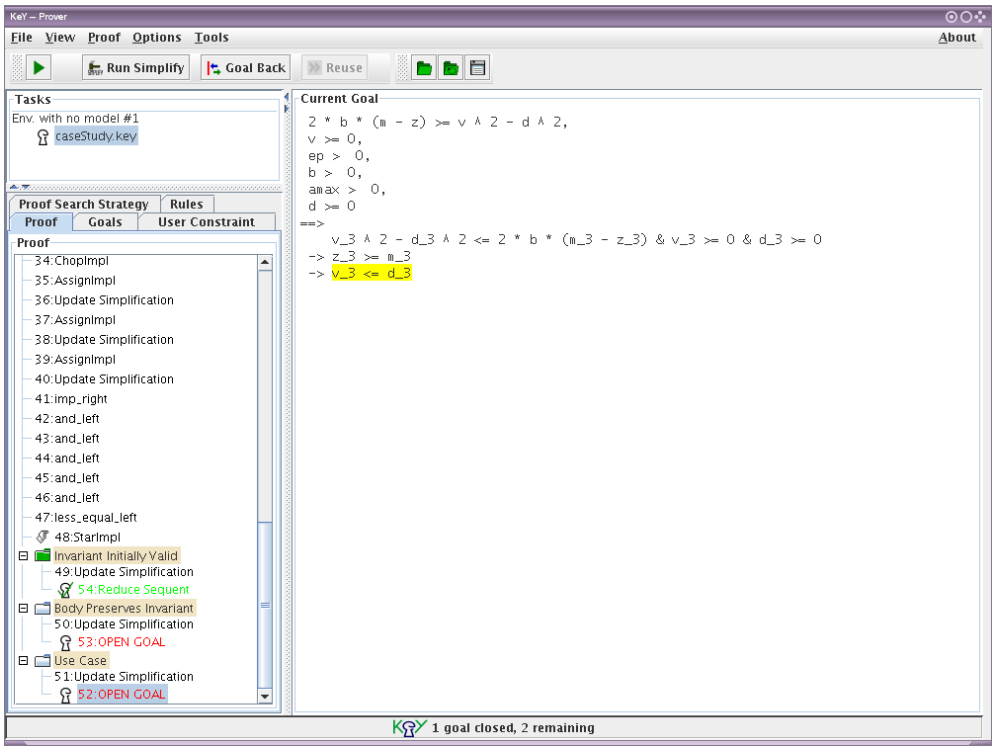


Figure 4.11: Snapshot of the KeY Prover

In KeY rules can be applied using the mouse. A single click on a formula (or a part of it) shows a context menu with all rules that are applicable on the current selection (see figures 4.12 and 4.13). Rules which have an `\assumes` clause can be applied either by clicking on the target formula and choosing the rule, or by dragging the assumption on the formula to be changed. If there is more than one rule applicable with this assumption and target, a menu is shown to let the user choose which rule should be applied.

Proofs can be saved either to complete them later or to document that a proof can be performed. Saving of proofs uses basically the same format as the problem files. Additionally, a Lisp like version of the current proof tree is saved. While loading the proof the rules are applied one after another again. This means that long calculations done by Mathematica have to be performed again.

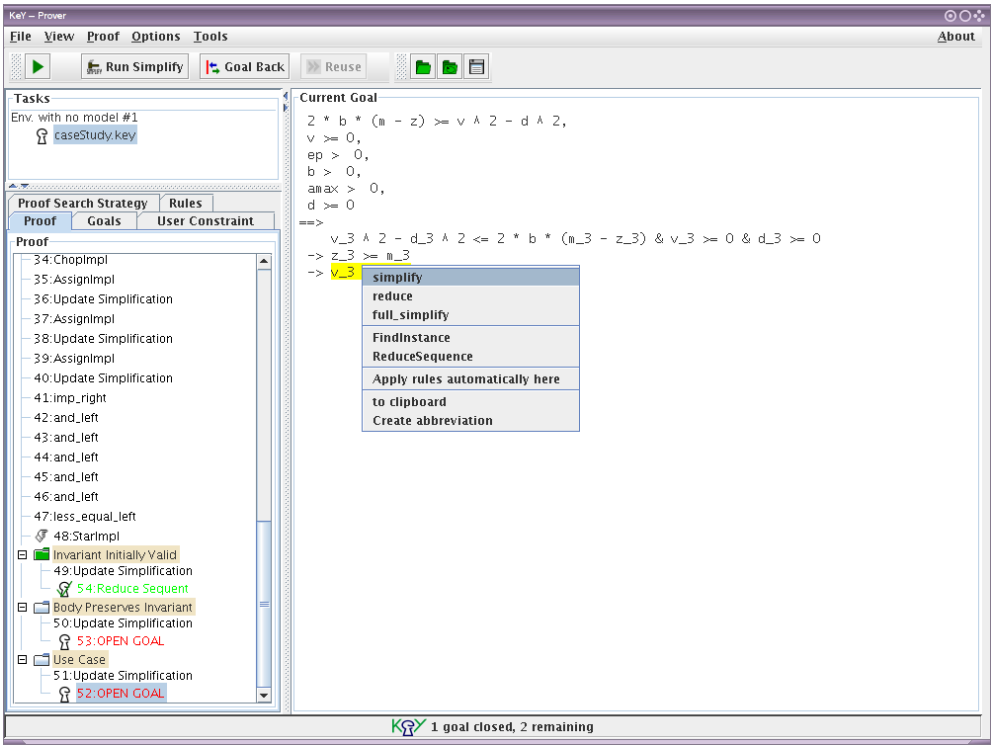


Figure 4.12: Snapshot of the KeY Prover with context menu opened on a subformula

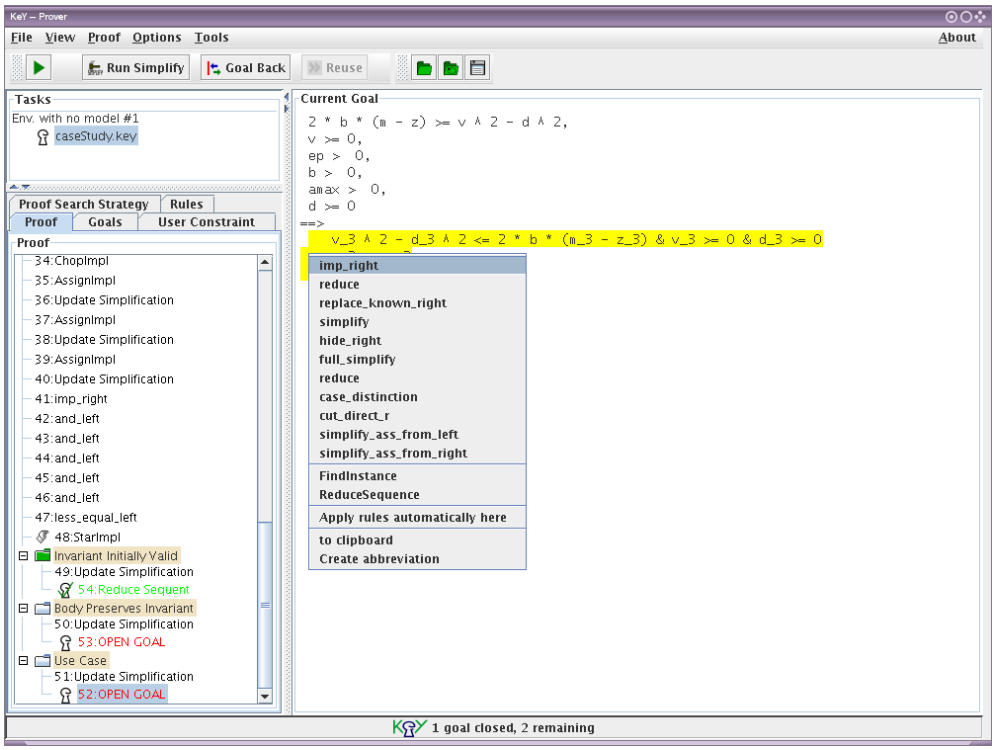


Figure 4.13: Snapshot of the KeY Prover with context menu opened on a complete formula

Abort Program As sometimes e.g. a `reduce` call could take very long, when given too much formulas or the wrong set of variables to reduce, it is necessary to be able to abort the calculation and proceed with the proof, i.e. try to close other goals, search for counter examples, or apply other rules to the current branch before trying to call `reduce` again. We could not integrate a button for canceling into the KeY GUI as rules are applied within the Swing event handling thread. This means that while a rule is applied the GUI events just queue up and are processed after the application of the rule is finished. As we want to cancel the current rule application, the event would be handled to late. We have realized this with an external program that links to the server and serves an abort button. A snapshot of the program is shown in figure 4.14.

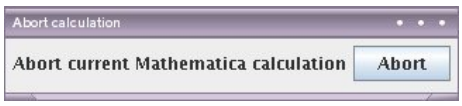


Figure 4.14: Snapshot of the Abort Program

Chapter 5

Case Study

5.1 Overview

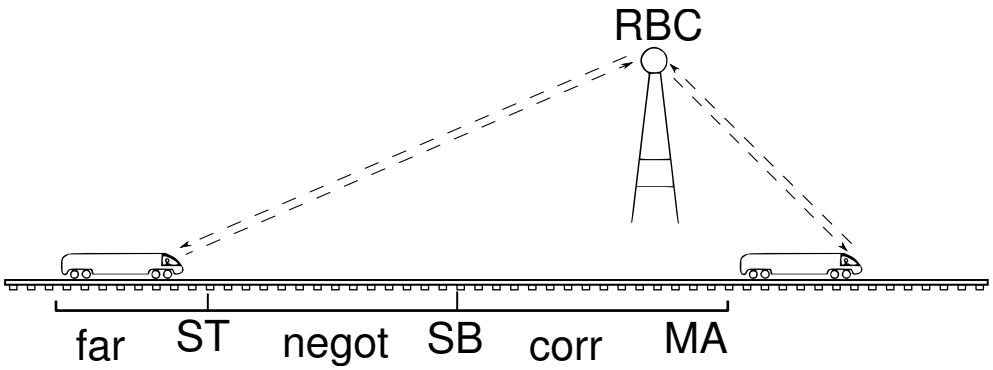


Figure 5.1: Communication between the train and the RBC

This case study is a simple version of the European Train Control System (ETCS) [ERT02] case study [DHO06, DMO+07]. It is based on the examples in [Pla07c] and [Pla07a]. The ETCS models a collision avoidance algorithm for trains. The idea is to give every train movement authorities, so that it can brake without colliding with another registered vehicle in case of emergency. A central controller instance, a Radio Block Controller (RBC), is responsible for managing these authorities.

The ETCS system is developed to create a system for collision avoidance that is used all over Europe. Currently there are many different systems used in the different nations and international trains have to implement all these systems, or exceptions have to be made for these trains.

In Germany currently a system is used that divides the railway system into fixed segments [JP04]. The idea is to achieve collision freedom by assigning a segment to at most one train at a time and keep a safety distance of at least one segment. This means that a train is blocking 2 to 3 segments when its on the rail. It always blocks the segment it is currently driving on, as well as the predecessor segment. A train which is close to the end of its current segment blocks 3 segments at once, as it has to request the successor segment before releasing the predecessor one to keep moving and preserve the safety distance. This is illustrated in figure 5.2.

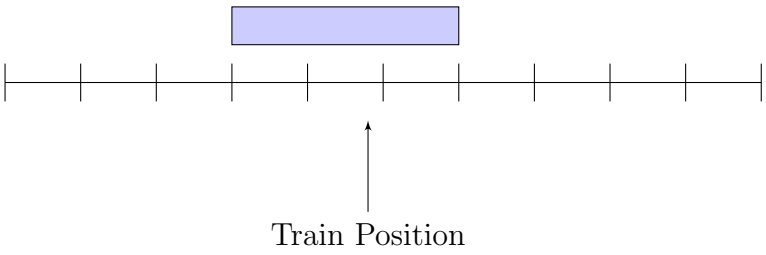


Figure 5.2: Train blocking 3 segments

A slow train driving behind might have to wait even if it is several minutes behind and would never reach the first train if it keeps on moving.

The ETCS introduces so called *moving blocks*. This means that there is a safety envelope around the train ensuring that the distance to any obstacles is large enough that it can brake down to a certain speed. This envelope is calculated depending on the maximal braking force of a train, the current velocity and the target speed.

The trains have to request movement authorities from the RBC which keeps track of the positions of all trains. This is similar to the requesting of segments in the current system. These movement authorities are used to assure that the train is slow enough if e.g. driving over bridges or railway crossings. Also they are used to achieve collision freedom.

When a train approaches the end of its movement authority, it passes a point ST (for "Start Talking"). This point marks the beginning of the communication with the RBC to get an extension of the movement authority. The RBC can either extend the movement authority or force the train to lower its speed to a certain amount. If there is no reaction of the RBC in time a point SB (for "Start Braking") is reached and braking is initialized.

The train has a desired speed it should keep on the rail. The speed supervision component of the train may not choose to accelerate if the speed is above this desired speed. On the other hand, the train is allowed to accelerate until this speed is reached.

The ETCS has a special focus on high speed trains. As the velocity of a train is a quadratic part of the distance needed to brake until a complete halt is reached it is significantly higher with growing velocity. For high speed trains the safety distance should be reduced. If the maximum braking force of a train is known, one can reduce the safety distance to the distance the rear train can brake down to the speed of the leading train, before colliding with it. The problem with this design is, that an external obstacle like e.g. a tree on the rail could increase the braking force of the leading train and collision could in the worst case not be avoided.

5.2 Formal Model

The RBC and the train are independent components that run in parallel. As we do not have a parallel operator we express the parallelism using the non-deterministic choice and repetition. This is possible as we model the RBC action to be instant. Time only passes while the train is moving.

We model the train using a differential system that formalizes the physical laws for movement. The first derivative of the position function z is the velocity v, the second derivative is the acceleration a. The time is modelled using a function with a constant derivative of 1.

System Parameters As we want to get a general result, we use parameters to describe the system.

- Constant parameters
  - a_{max} is the maximum acceleration
  - b is the maximum acceleration that can be used for braking
  - ε is the maximum response time of the train speed supervision
- System variables
  - z is the current train position
  - v describes the velocity
  - a is the acceleration
  - t is a clock used for measuring the length of driving phases
- Control parameters
  - SB is the point where the train speed supervision needs to force braking
  - ST is the point where the communication with the RBC is started
- RBC variables
  - m is the end of movement authority
  - m_{old} is used to store the previous end of movement authority
  - d is the maximum value allowed for the velocity of the train at the end of movement authority
  - d_{old} is, like m_{old}, used to store the previous value of d.
  - v_{des} provides a desired speed for the train in normal operation
- State model
  - *state* stores the current state of the train
  - *drive*, *brake* are used as enumeration values for the *state*.

Phases of the Train Control The train always passes the same stages. First it polls its current position and recalculates the values for SB and ST. Afterwards, if the point ST is passed, a communication with the RBC may be initialized. Next the speed control chooses the acceleration depending on the relation between the current and the desired speed or chooses to brake if the RBC demands it. Now the speed supervision is invoked checking whether the point SB is passed, which would cause immediate recovery. For verifying safety properties it is sufficient to choose an acceleration between the maximum braking force and the maximum acceleration when the speed is below the desired speed and between the maximum braking force and zero otherwise. Note that for liveness a strict positive acceleration in the first case would be necessary. The last stage is the driving stage, which is performed for a maximum of ε time units.

The train behavior is illustrated in figure 5.3.

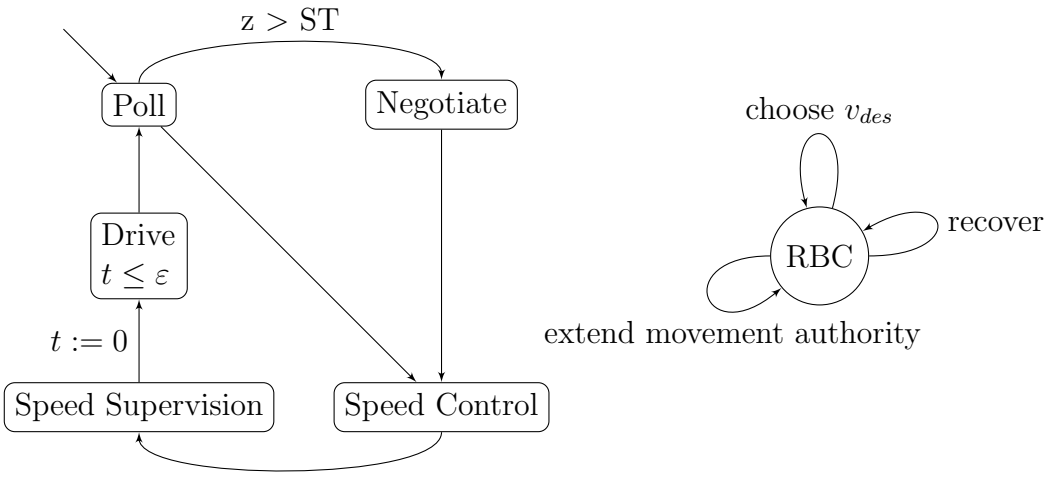


Figure 5.3: Train model as automata

The RBC Model In our model the RBC is a rather simple controller as it needs to control only one train. It has three possibilities to react. It can either choose to extend the current movement authority, force braking or choose a new desired speed for the train.

If the movement authority is extended the extension is provided by new values for m and d. The new value for d is depending on the new position of m. It has to be possible to brake from the previous target to speed down to the new one within the extension range, as the train guarantees that it can brake down to the previous target speed before it reaches the previous end of the movement authority. Now the RBC could choose just at this moment to extend the authority. This is the worst case as the previous target speed is the highest speed that could be obtained at the closest position to the new end of movement authority. If no extension is given the train keeps braking until it comes to a complete halt.

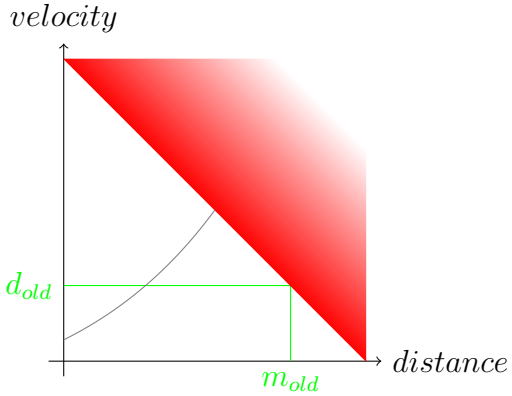
This RBC can be refined without endanger the safety of the train system. In our model the desired speed may change without any constraints. Constraints could

be added for e.g. map desired speeds to certain track sections. Also the RBC may choose to recover independent on the current situation. An extension could be made that restrict this choice to the cases in which recovery is necessary. For the case that the movement authority is extended it is necessary that the train is able to brake from its current speed and current position down to the new target speed before reaching the end of the extended movement authority. Every refinement that satisfy this constraint is thinkable. For example it would be possible to add a second train that determines the current movement authority instead of choosing it randomly within the braking constraints.

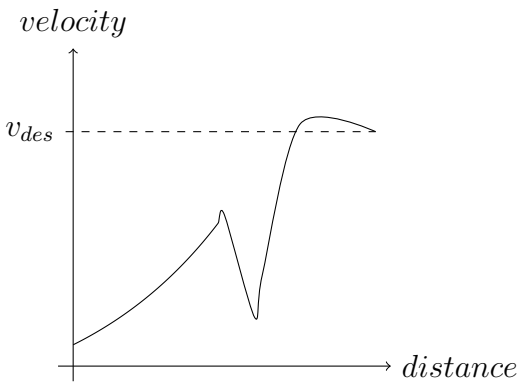
The possible choices for our RBC model are illustrated in figure 5.3.

Compared to the examples in [Pla07c] and [Pla07a] the RBC is modelled more explicitly. We also extended the model by the introduction of the target and the desired speed. The train behavior is also more realistic in our model, as the acceleration of the train is not only determined by a maximum deceleration, zero and a maximum acceleration but by the interval between the maximum deceleration and the maximum acceleration.

Example 12. *In the following diagram the area of valid extensions of the movement authority is marked red. It extends to left and upwards. The gray line is a possible movement of the train. The train could move below the red line, but the minimum slope of the trains speed is given by the slope of the border line of the area of valid extensions.*



To get an idea of the possible train behavior have a look at the next plot.



$$
\begin{array}{lcl}
\text{system} & : & \big(\text{poll};\ (\text{negot} \cup (\text{speedControl};\ \text{speedSup};\ \text{move}))\big)^* \\
\text{init} & : & drive := 0;\ brake := 1 \\
\text{poll} & : & SB := \frac{v^2-d^2}{2b} + \big(\frac{a_{max}}{b} + 1\big)\big(\frac{a_{max}}{2}\varepsilon^2 + \varepsilon v\big);\ ST := * \\
\text{negot} & : & (?m - z > ST) \cup (?m - z \leq ST;\ \text{rbc}) \\
\text{rbc} & : & (vdes := *;\ ?vdes > 0) \cup (state := brake) \\
 & & \cup \big(d_{old} := d;\ m_{old} := m;\ m := *;\ d := *; \\
 & & \quad ?d \geq 0 \wedge d_{old}^2 - d^2 \leq 2b(m - m_{old})\big) \\
\text{speedControl} & : & \Big(?state = brake;\ \big((?v > 0;\ a := -b) \cup (?v = 0;\ a := 0)\big)\Big) \\
 & & \cup \Big(?state = drive; \\
 & & \quad \big((?v \leq v_{des};\ a := *;\ ? - b \leq a \leq a_{max}) \\
 & & \quad \cup (?v \geq v_{des};\ a := *;\ ?0 > a \geq -b)\big)\Big) \\
\text{speedSup} & : & \Big(?m - z \leq SB;\ \big((?v > 0;\ a := -b) \cup (?v = 0;\ a := 0)\big)\Big) \\
 & & \cup (?m - z > SB)) \\
\text{move} & : & t := 0;\ \{\dot{z} = v, \dot{v} = a, \dot{t} = 1, (v \geq 0 \wedge t \leq \varepsilon)\}
\end{array}
$$

Table 5.1: System specification

Specification In table 5.1 the formal model of the system is shown. The system process first polls its current speed and target speed to calculate the size of the safety envelope.

At the point SB the train has to start braking. The distance between SB and the end of the movement authority m is given by the minimum braking distance needed to brake from the current speed v to the target speed d and additionally the distance the train covers within one controller cycle with the resulting braking distance extension for the maximal speed that the train could reach within that cycle. The minimum braking distance can be calculated by $\frac{v^2-d^2}{2b}$. The train can accelerate with a maximum acceleration of a_{max} and brake with a maximum deceleration of b. Assuming that the train accelerates with its maximum acceleration it covers a distance of $\big(\frac{a_{max}}{2}\varepsilon^2 + \varepsilon v\big)$ within ε time units. As the train needs to brake down from the speed it could reach within this time, we need to add the ratio between the maximum acceleration and the maximum deceleration for this distance as well. In combination this means that we need to extend the safety distance by $\big(\frac{a_{max}}{b} + 1\big)\big(\frac{a_{max}}{2}\varepsilon^2 + \varepsilon v\big)$. This constraints is taken from [DMO+07].

After the polling it is non-deterministically chosen if there is a negotiation phase or if the train is moving. While the negotiation phase no time does elapse. The negotiation phase simply checks whether the train has passed the point ST, which marks the point from which the train may communicate with the RBC, or not. If the train has passed the point the RBC is invoked. The RBC can either choose to adjust the desired speed, force braking or extend the movement authority. An extension has to satisfy two constraints. First, the target speed must be greater as or equal to zero. Second, it must be possible to brake down from the previous target speed to the new one within the extension distance. For

the case that the case that the new end of the movement authority is closer to the train than the previous one, the target speed has to increase such that the train can brake from the new target speed down to the previous one within the "extension" distance.

In the other case, the case in which the train is moving and time can pass, the speed control is invoked first. The speed control initiates braking if the train is in state *brake* and it is still moving. In the state *drive* it regulates the speed depending on a desired speed. If the speed is less or equal to the desired speed the acceleration is chosen from the interval $[-b, a_{max}]$, otherwise it is chosen from $[-b, 0[$. The next component to invoked is teh speed supervision. This is a simple controller that monitors the distance to the end of the movement authority and compares it to the point SB. If SB is passed braking with maximum deceleration is forced. Last instance is the moving stage. Here the train moves for a maximum of ε time units with the chosen acceleration from the current speed and the current position. This is modelled with the differential equation system $\{\dot{z} = v, \dot{v} = a, \dot{t} = 1, (v \geq 0 \wedge t \leq \varepsilon)\}$. For this differential equation system it is invariant that the velocity of the train is a non-negative real number.

We want to verify that the trains velocity is always sound with respect to the distance profile chosen by the RBC. To assure this, we need to verify that the train has a maximum speed of d at the point m and beyond this position if no extension of the movement authority is granted. Therefore we need to proof that for all traces of the system the following postcondition holds.

$$(m \geq z \rightarrow v \leq d)) \tag{5.1}$$

We assume that the reaction time of the train is greater zero. The same assumption is made about the maximum deceleration. We further assume that the initial velocity is greater or equal two zero, i.e. the train is not moving backward and the initial target speed is greater or equal to zero as well.

From an altered train model in which the train always chooses it brake with the maximum braking force we can derive that initially either the distance to the end of the movement authority is large enough to brake down from the current speed to the target speed, or the velocity of the train is below the target speed already. This is expressed by the following constraint.

$$v^2 - d^2 \leq 2b(m - z) \tag{5.2}$$

The constraint says that there is enough room to brake from the current speed down to the target speed before reaching the end of the movement authority. $\frac{v^2-d^2}{2b}$ is the distance needed to brake from a speed v down to a speed d with a deceleration of b. This approach to find this constraint is taken from [Pla07c].

The same pattern of constraint can be found in the RBC component in the case, where it is extending the movement authority. It is necessary that the train can brake from the previous target speed down to the new one on the extension distance. It would be sufficient if it could brake down from its current speed to the new target speed from its current position to the new end of the movement authority, but this would require the RBC to know the exact position of the train

and the communication overhead would be too big. As the train asserts that its movement complies with the previous movement authority, the RBC knows the maximum speed of the train at the end of the movement authority and can use it for its calculations easily.

Our proof obligation is

$$[init]\Big(\big(\varepsilon > 0 \land b > 0 \land v^2 - d^2 \leq 2b(m-z) \land v \geq 0 \land d \geq 0\big) \\ \rightarrow [system](m \geq z \rightarrow v \leq d)\Big) \quad (5.3)$$

The modality $[init]$ is only used for initializing our enumeration types for the train controller states.

5.3 Verification

For proving that formula (5.3) is valid, we need a system invariant that is strong enough to imply all necessary properties. In this section we present a proof draft that is using the following invariant:

$$v^2 - d^2 \leq 2b(m-z) \land d \geq 0 \land v \geq 0 \quad (5.4)$$

A draft of the first proof steps is presented below.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{inv} \vdash \forall ST_2[\text{negot}]\text{inv} \qquad \text{inv} \vdash \{SB\}[\text{speed};\text{move}]\text{inv}}{\text{inv} \vdash \forall ST_2[\text{negot}]\text{inv} \land \{SB\}[\text{speed};\text{move}]\text{inv})}\,\text{R7}}{\text{inv} \vdash \{SB\}\forall ST_2(([\text{negot}]\text{inv}) \land [\text{speed};\text{move}]\text{inv})}}{\text{inv} \vdash [\text{poll};(\text{negot} \cup (\text{speed};\text{move})]\text{inv}}}{\dfrac{\dfrac{\overline{\text{initial} \vdash \text{post}}\,* \qquad \text{inv} \vdash [\text{system}]\text{inv} \qquad \overline{\text{inv} \vdash \text{post}}\,*}{\text{initial} \vdash [\text{system}]\text{post}}\,\text{R29}}{\vdash \text{proofObligation}}}$$

We use the following abbreviations:

proofObligation stands for equation (5.3)

initial abbreviates the constraints for the initial state

post is the postcondition (equation (5.1))

inv is the system invariant (equation (5.4))

$\{SB\}$ is the update that sets the new value for SB

speed is an abbreviation for $speedControl; speedSup$

On the right side of the sequent are the hidden formulas $b > 0$ and $ep > 0$.

The proof proceeds as follows. First the initial assignments are converted into updates. Afterwards the sequent is sorted using the rules for propositional formulas. This results in the following sequent:

$$initial \vdash [system]post$$

If we eliminate some abbreviations we get:

$$\varepsilon > 0, b > 0, v^2 - d^2 \leq 2b(m - z), v \geq 0, d \geq 0 \quad \vdash \quad [system](m \geq z \rightarrow v \leq d)$$

Now the invariant rule R29 is applied. This leads to three new proof goals. The first is to show that the invariant is initially valid.

$$\varepsilon > 0, b > 0, v^2 - d^2 \leq 2b(m - z), v \geq 0, d \geq 0 \vdash v^2 - d^2 \leq 2b(m - z) \wedge d \geq 0 \wedge v \geq 0$$

This goal can be closed by applying `ReduceSequence` and let Mathematica find out that this is a tautology or by applying rules for propositional logic and close the three resulting branches.

The other two open goals are the following. We have to show that the invariant is preserved by the loop body and we have to show that the invariant is strong enough to imply the postcondition.

The latter one is simple again. The sequent looks like this (we removed the formulas that cannot lead to closure of this goal):

$$\varepsilon > 0, b > 0 \quad \vdash \quad (v^2 - d^2 \leq 2b(m - z) \wedge d \geq 0 \wedge v \geq 0) \rightarrow (m \geq z \rightarrow v \leq d)$$

Applying the rule R3, two times R6, and R3 again we get:

$$\varepsilon > 0, b > 0, v^2 - d^2 \leq 2b(m - z), d \geq 0, v \geq 0, m \geq z \vdash v \leq d$$

This sequent can be reduced to `true` by Mathematica.

The last open goal, the induction step is not that simple to prove. This goal splits into two major proof obligations. This split is caused by the nondeterministic choice between the RBC behavior and the train movement. We have to show that the system is safe if the RBC performs actions, as well if the train control is working and the train is potentially moving.

5.3.1 RBC Behavior

Applying our proof strategy, it is possible to show automatically that the RBC behavior is not harmful to the safety of the system. This is the case in the proof draft where we have to show that

$$\text{inv} \vdash \{SB\}\forall ST_2[\text{negot}]\text{inv}$$

holds. As SB does not occur neither in inv nor in negot, we just drop the update.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{*}{\text{inv}, \Delta > sk \vdash \text{inv}}\text{R11}}{\text{inv} \vdash [(?\Delta > sk)]\text{inv}} \qquad \dfrac{}{\text{inv}, sk \geq \Delta \vdash [\text{rbc}]\text{inv}}}{\text{inv} \vdash [(?\Delta > sk)]\text{inv} \wedge [?\Delta \leq sk;\ \text{rbc}]\text{inv}}\text{R7}}{\text{inv} \vdash [(?\Delta > sk) \cup (?\Delta \leq sk;\ \text{rbc})]\text{inv}}\text{R21}}{\text{inv} \vdash \forall ST_2[(?\Delta > ST_2) \cup (?\Delta \leq ST_2;\ \text{rbc})]\text{inv}}\text{R12}}{\text{inv} \vdash \forall ST_2[\text{negot}]\text{inv}}$$

Δ is used to abbreviate the difference between the end of the movement authority m and the current position z

The quantifier over ST_2 is handled using Skolemization. Afterwards the proof splits depending on if the point ST has been passed. If the train has not passed it yet. The modality does not alter any variables, thus the sequent becomes true and this branch can be closed. If the train has passed the point ST, the RBC is invoked.

$$\dfrac{\dfrac{\dfrac{*}{\text{inv} \vdash [(vdes := *;\ ?vdes > 0) \cup (state := brake)]\text{inv}} \qquad \text{inv} \vdash [\text{extend}]\text{inv}}{\text{inv}, sk \geq \Delta \vdash [(vdes := *;\ ?vdes > 0) \cup (state := brake) \cup \text{extend}]\text{inv}}}{\text{inv}, sk \geq \Delta \vdash [\text{rbc}]\text{inv}}$$

extend is used to abbreviate

$$\big(d_{old} := d;\ m_{old} := m;\ m := *;\ d := *; ?d \geq 0 \wedge d_{old}^2 - d^2 \leq 2b(m - m_{old})\big)$$

As the variable sk does not occur in the RBC specification we can hide the constraints. The RBC model has 3 possible choices. It can either choose a new desired speed, force braking or extend the movement authority. The first two cases can be easily handled as they do not alter variables occurring in the invariant. The third case does alter the variables d and m that are part of the invariant.

$$\dfrac{\text{inv} \vdash \forall m_2 \forall d_2 (d_2 \geq 0 \wedge d^2 - d_2^2 \leq 2b(m_2 - m) \rightarrow \text{invS})}{\text{inv} \vdash [\text{extend}]\text{inv}}$$

invS is the abbreviation for $\text{inv}[m \mapsto m_2][d \mapsto d_2]$

After Skolemization this goal can be split into three branches, one for each part of the invariant. The branches are (hiding the unnecessary formulas).

The first goal is:

$$v^2 - d^2 \leq 2b(m - z) \wedge d \geq 0 \wedge v \geq 0 \vdash v \geq 0$$

This is obviously be true. The second goal is:

$$\text{inv}, sk_d \geq 0 \vdash sk_d \geq 0$$

This as well is true and can be closed. The symbol sk_d is the Skolem symbol for d_2. The third goal is:

$$\text{inv}, d^2 - sk_d^2 \leq 2b(sk_m - m) \vdash v^2 - sk_d^2 \leq 2b(sk_m - z)$$

Here we use sk_m as the name for the Skolem symbol introduced for the quantified symbol m_2.

This goal can be closed as well. The invariant says that we can brake down from the position z and the speed v down to the speed d. The other constraint on the left side says that we can brake down from the speed d to the speed sk_d on a distance of $sk_m - m$. Putting this two constraints together we can derive that the train can brake from a speed v down to a speed sk_d on a distance of $sk_m - z$ which is the succeedent of the sequent.

5.3.2 Train Controller

A draft of the first steps in the branch aiming at the verification of the train controller is presented below.

$$\dfrac{\dfrac{\dfrac{\text{inv} \vdash \{SB\}[doBrake]\varphi \qquad \text{inv} \vdash \{SB\}[doDrive]\varphi}{\text{inv} \vdash \{SB\}[(?state = brake; doBrake) \cup (?state = drive; doDrive)]\varphi}}{\text{inv} \vdash \{SB\}[\text{speedControl}][\text{speedSup}][\text{move}]\text{inv}}}{\text{inv} \vdash \{SB\}[\text{speed}; \text{move}]\text{inv}}$$

doBrake is the abbreviation for $\Big((?v > 0;\ a := -b) \cup (?v = 0;\ a := 0)\Big)\Big)$

doDrive abbreviates the acceleration choice depending on the desired speed

φ is used to abbreviate [speedSup][move]inv

For the case in which the speedControl behavior is determined by doBrake, we can close the branch easily using the braking constraint (5.2). This constraint guarantees that the distance to the end of the movement authority is always large enough to brake down to the target speed.

For the other case, the case where doDrive is executed, it is more difficult to show that no unsafe state can be reached.

Here the proof splits as the behavior of the speed control depends on the relation between the current speed and the desired speed. If the train passed the point SB,

we can hide the formula for SB. This is possible as in this case the acceleration is set to the maximum deceleration by the speed supervision component. We can close this branch using mainly the knowledge from the braking constraint (5.2).

For the other case we have to show that the knowledge that the train has not passed the point SB is enough for asserting that the system is still safe. In this case the speed supervision component does not react and two cases result from the speed control component again.

The two goals are:

$$\text{inv}, \Delta > SB, v_{des} \geq v \vdash \forall a(-b \leq a \leq a_{max} \rightarrow [\text{move}]\text{inv})$$

$$\text{inv}, \Delta > SB, v \geq v_{des} \vdash \forall a(-b \leq a < 0 \rightarrow [\text{move}]\text{inv})$$

This can be proven again using Skolemization in the first place and eliminating the quantifiers for the time and for the acceleration at once.

The two sequents we have to prove valid are:

$$\begin{aligned}\big(0 \leq t \leq \varepsilon \wedge a + \frac{v}{t} \geq 0 \wedge m - z > \frac{v^2 - d^2}{2b} + (\frac{a_{max}}{b} + 1)\big(\frac{a_{max}}{2}\varepsilon^2 + \varepsilon v\big) \\ \wedge -b \leq a \leq a_{max} \wedge 2b(m-z) \geq v^2 - d^2 \wedge d \geq 0 \wedge v \geq 0 \wedge \varepsilon \geq 0 \\ \wedge b \geq 0 \wedge a_{max} \geq 0\big) \rightarrow (at + v)^2 + 2b(\frac{a}{2}t^2 + 2vt + 2z - 2m) \leq d^2\end{aligned}$$

and

$$\begin{aligned}\big(0 \leq t \leq \varepsilon \wedge a + \frac{v}{t} \geq 0 \wedge m - z > \frac{v^2 - d^2}{2b} + (\frac{a_{max}}{b} + 1)\big(\frac{a_{max}}{2}\varepsilon^2 + \varepsilon v\big) \\ \wedge -b \leq a < 0 \wedge 2b(m-z) \geq v^2 - d^2 \wedge d \geq 0 \wedge v \geq 0 \wedge \varepsilon \geq 0 \\ \wedge b \geq 0 \wedge a_{max} \geq 0\big) \rightarrow (at + v)^2 + 2b(\frac{a}{2}t^2 + 2vt + 2z - 2m) \leq d^2\end{aligned}$$

These sequents only differ in the constraints for the current acceleration. It can either be $a \in [-b, a_{max}]$ or $a \in [-b, 0[$. In both cases the constraint on the right side of the sequent specifies that the braking constraint still holds after performing a accelerated movement for t time units with $t \in [0, \varepsilon]$ from the speed v with the acceleration a. As the constraint for SB specifies that it is always safe to perform an accelerated movement with the maximum acceleration from the current velocity for ε time units, we can close both of these goals.

While performing the verification of the case study, we have observed that there are cases where Mathematica needs significantly more time to eliminate one variable instead of two.

Example 13. *Calling* `reduce` *on the following formula with either the target set* $\{t\}$ *or* $\{a\}$ *does not produce an result within 2 hours, whereas the target set* $\{t, a\}$ *lets Mathematica simplify this formula to true within less than 2 minutes. Another important point is the order of the reintroduced quantifiers. Adding the*

quantifier prefix as $\forall a \forall t$ leads to a significant performance gain in comparison with the case where we add it as $\forall t \forall a$. This is a surprising result, as the prefixes are equivalent.

$$\begin{aligned}\big(0 \leq t \leq \varepsilon \wedge a + \frac{v}{t} \geq 0 \wedge m - z > \frac{v^2 - d^2}{2b} + (\frac{a_{max}}{b} + 1)\big(\frac{a_{max}}{2}\varepsilon^2 + \varepsilon v\big) \\ \wedge 0 \leq a \leq a_{max} \wedge 2b(m-z) \geq v^2 - d^2 \wedge d \geq 0 \wedge v \geq 0 \wedge v_{des} \geq 0 \wedge \varepsilon \geq 0 \\ \wedge b \geq 0 \wedge a_{max} \geq 0\big) \rightarrow (at+v)^2 + 2b(\frac{a}{2}t^2 + 2vt + 2z - 2m) \leq d^2\end{aligned}$$

Statistics To give an idea how long it takes to proof the case study in our tool we present a few statistics at this point. The whole proof of the case study can be performed with 266 rule applications. These rule applications resulted in 17 branches and we performed 56 interactive steps. The proof can be performed automatically to the point where the invariant rule has to be applied. After this step the user should hide the unnecessary formulas in the induction step branch. Now the proof can be performed automatically again until there are 3 open goals left. Up to this point 9 goals could already be closed automatically. The goals that cannot be closed automatically within certain time bounds contain huge formulas containing many quantifiers. One of this targets is the case the train is in the state *brake* in the other cases the train is in the case *drive*. The case where the train is already in the state *brake* can be closed with by performing a few steps manually. It is necessary to hide the constraint for SB as it is unnecessary for the closure of this branch but causes the Mathematica calculation to take significantly longer. With another 12 user interactions it is possible to reduce the open goals to the cases where the train is driving and has chosen either an acceleration between $-b$ and 0 or up to a_{max}. These branches can be closed with about 41 user interactions. Most of these are applying basic rules to be able to hide formulas that only consume calculation time. The two user interactions are of capital importance. These are the two applications of `EliminateQuantifierWithContext` with a given set of variables (see the example above). Overall it takes about ten minutes to perform this proof when trying to let the prover run in automatic mode most of the time. About five minutes are calculation time.

Chapter 6

Related Work

In this chapter we compare our approach of the verification of hybrid systems with other projects to give an idea of the context of our work.

Theorem Provers Implemented in Mathematica

There are two approaches to implemented a theorem prover directly into Mathematica. One is Analytica [CZ92] the other is Theorema [BJK+97].

Analytica was developed for Mathematica 1.2. It is a theorem prover for the 19th century mathematics. In 2003 an approach was made to port Analytica to the new version of Mathematica (version 5). This approach is called Analytica 2 [CKOS03]. Major code cleanups have been made and Analytica 2 can prove the same theorems as its predecessor. A major problem with this prover is the lack of documentation.

Theorema delivers an approach for the integration of a computer algebra system with a theorem prover which is different from Analytica. Theorema is a theorem prover written in the programming language provided by Mathematica but it does not rely on the algebraic algorithms library [BJK+97]. Theorema provides various natural deduction theorem provers for special classes of higher-order logic as well as first-order logic. The authors decided to use the user interface provided by Mathematica. The project aims at the combination of functor style programming with proving. A functor is a construct used in category theory. In Theorema functors are used to, given a domain, define new domains and transport knowledge to these new domains [BJK+97].

Theorema does not aim at the verification of programs, but an experimental approach has been developed in 2004 [JKP04].

As both approaches do not aim at the verification of hybrid systems and the logics used are not similar to $\mathsf{d}\mathcal{L}$, they are far from our needs.

Model Checkers for Hybrid Systems

Model checker [CGP99] for hybrid systems aim at verification by exploring the state space of the system. HyTech [HHWT97] is a familiar model checker for this purpose. It is capable of checking linear hybrid automata [HHWT97, ACHH92]. A hybrid automaton is a finite automaton with a finite number of real-valued variables. These variables may change continuously along differential equations

or within the range defined by differential inequalities. Therefore hybrid automata are a generalization of timed automaton [AD94], where the value of the variables (clocks) change with the constant slope of 1. A linear hybrid automaton is a hybrid automaton where all predicates are convex, i.e. they are conjunctions of inequalities over a set of variables. HyTech works fully automatically and checks a given system for correctness with respect to a given requirement. The requirements are specified in a temporal logic. As the state space of linear hybrid automata is infinite, symbolic methods are used to handle it. HyTech needs concrete numbers for most of the parameters. With our approach it is possible to derive parameter constraints and abstract from concrete numbers.

Another model checker for hybrid systems is PHAVer [Fre05]. PHAVer also aims at the verification of safety properties of linear hybrid automata. It extends the automata model by allowing affine dynamics that are handled by on-the-fly overapproximation.

André Platzer presented in [Pla07c] a translation from linear hybrid automaton into $\mathsf{d}\mathcal{L}$, thus in general it should be possible to verify the same examples in KeY as in HyTech and PHAVer.

Integration of Maple and PVS

Adams, Dunstan et al. provide a description how Maple [CGG83] and PVS [ORS92] were integrated to prove certain properties of mathematical functions in [ADG+01]. Maple is a computer algebra system like Mathematica and PVS is a theorem prover for higher-order logic with support for predicate and dependent sub-typing. Their approach is giving the user access to the theorem prover from the Maple working perspective, thus one can prove e.g. if a function is continuous using PVS. This information may be necessary to verify the correctness of the results supplied by Maple. They integrated a special PVS library for proving properties like continuity or convergence of transcendental functions [Got00].

The main difference in their approach is that they only use the theorem prover to verify properties like continuity of functions, whereas we use the theorem prover as our main component. Furthermore, they utilize the CAS as user interface, because they employ the theorem prover for supporting the user while performing mathematical calculations. Another difference is that they use the theorem prover to check the results of the CAS, whereas we rely on the results of the black-box application Mathematica. It may be noted that there are also approaches to verify Java programs using PVS [Hui01], but they use higher-order logic instead of dynamic logic, thus it is not as close as KeY to our needs.

Verification of Hybrid Systems using PVS

In [ÁMSH01b] Erika Ábrahám-Mumm, Ulrich Hannemann and Martin Steffen presented a deductive proof method for the verification of hybrid systems. For

the representation of hybrid systems they use hybrid automata [ACHH92] with step semantics. The step semantics defines that a run of a hybrid automaton is the union of discrete state changes and time steps, i.e. the transitions of the hybrid system where time passes and the variables may change. The verification of properties is based on the inductive assertion method [Flo67]. They use inductive assertion networks for the representation of system invariants. An assertion on a location of the hybrid automaton is a boolean predicate over the variables of the automaton. An assertion network assigns assertions to locations of the hybrid automaton [ÁMSH01a]. An assertion network is called inductive if it holds for all initial states of the hybrid automaton and is preserved under the transition relation [ÁMSH01a].

The authors have integrated a model for hybrid automata, as well as for its semantics into PVS. The user has to provide the automaton and can use the description of the system as hybrid automaton. They prove that the given assertion network is invariant and inductive for the given automata by projecting the automaton on its semantics and applying proof rules provided by PVS. This way they can verify safety properties of hybrid systems.

They also integrated a proof rule for reducing the complexity of verification of parallel systems into PVS.

The main difference to our approach is the input language. They use hybrid automaton and higher-order logic, whereas we use a dynamic logic ($\mathsf{d}\mathcal{L}$). Also they use the higher-order facility of PVS to handle arithmetics, whereas we integrated the prover with an external program for solving the arithmetics. The handling of the parallel composition of hybrid systems is another advantage of their implementation, as at the moment it is necessary in our implementation to compute the parallel composition of the system components by hand. Another difference is the fact that they translate the automaton into higher-order logic in one step, whereas we inductively process the input formula.

STeP

STeP [BBC+96] is a combined system of a deductive prover and a model checker. The basic idea is to model a system using transition systems and verify it on the basis of verification rules and verification diagrams. It reduces temporal properties to first-order conditions, thus the deductive prover component can handle them. The verification diagrams are designed for proof monitoring of the user. STeP integrates a automatic prover, an interactive prover and a model checker. The interactive theorem prover is a Gentzen-style first-order prover [BBC+96, Gal87]. STeP also uses certain techniques for automatic detection of local, linear or polyhedral invariants [BBM97].

In [MS98] Zohar Manna and Henny B. Sipma presented a way to verify safety properties of hybrid systems using STeP. They use Phase Transition Systems to model the systems. A Phase Transition System is a transition system extended by activities [MS98]. These activities are differential inclusions of variables. The functions used to specify the bounds of the variable evolution may not contain

variables that are bounded by differential inclusions themselves. Trace semantics are used to verify safety properties of these systems. Using trace semantics means that the logic can specify properties that hold in any state of the system. Temporal properties are reduced to a set of first-order verification conditions by an invariant rule. An extended version of $\mathsf{d}\mathcal{L}$ is presented in [Pla07a] that also uses trace semantics for the verification of temporal properties.

In [SSM04] the methods for finding invariants in hybrid systems are described. The authors reduce the problem of finding invariants to a constraint solving problem. REDLOG [DS97] is used to handle non-linear clauses. In [MS98] a plan is illustrated to integrate STeP with REDLOG. We have found no indices that this has ever happened.

One advantage of STeP are the procedures for finding invariants automatically. The integration of REDLOG would be similar to our integration of Mathematica and one could think of integrating REDLOG with KeY as well. The trace semantics make it possible to use modalities to formalize propositions about every state of a system, whereas in the current implementation, we cannot assert that a solution to a differential equations is not totally out of bounds while evolving, i.e. we can only make propositions about the initial values and the values after the continuous evolution is performed. As said earlier theories have been developed to overcome this [Pla07a] and one can think of extending KeY, thus it can check these properties too.

Comparison between STeP and the PVS approach

We have remarked that the approaches for the verification of hybrid systems using STeP [MS98] and PVS [ÁMSH01b] have some properties in common. Both approaches base on a transition system description of the hybrid system. The phase transition systems are, beside the restrictions of the differential inclusions, very much like hybrid automata. Both approaches use a translation step from the input to easier proof goals formulated in a logic. They both supply an invariant rule to enable the user to proof safety properties. No rules are supplied that aim at the verification of liveness properties.

Compared to our approach, the most obvious difference is the translation to a logic level at once. As we use a dynamic logic that is used to specify the system as well as the property to proof, we can perform the proof inductively. Also it is possible to use our calculus to proof liveness properties, but as our implementation lacks of general handling of existential quantifiers on the right side of the sequent we must admit that this makes verification of liveness properties very difficult and in many cases impossible.

Chapter 7

Conclusions

Results

This thesis aims at the development of a software system for the computer aided verification of hybrid systems. We use the logic $\mathsf{d}\mathcal{L}$ [Pla07c, Pla07b, Pla07a] as a semantically well-founded specification language to describe both hybrid system behaviours and correctness statements. For the deductive verification of hybrid systems, we implement the $\mathsf{d}\mathcal{L}$ calculus in the theorem prover KeY [BHS07].

In chapter 2 we recapitulate the definition of the logic $\mathsf{d}\mathcal{L}$ as well as a sound sequent calculus that can be used to perform proofs on a syntactical level. We have discussed different ways to handle quantifiers. For the universal fragment of $\mathsf{d}\mathcal{L}$, that is, the fragment containing only universal quantifiers on the right and existential quantifiers on the left side of the sequent, we have shown that Skolemization [Fit96] can be used. This way we can avoid a side deduction in these cases. Existential quantifiers on the right side and universal quantifiers on the left side of the sequent could be handled using side deduction [Pla07c] as a complementary technique. Further we have presented our design for a verification tool based on this calculus in chapter 3 and we described the implementation using the theorem prover KeY and the computer algebra system Mathematica [Wol03] in chapter 4.

With the current implementation it is not possible to check formulas outside the universal fragment like $\vdash \exists x[\alpha^*]\varphi$. Still most safety properties depend on universal quantification of the system parameters within a certain range, so that there often is no need for an existential quantifier. If the existential quantifier does not occur in front of a loop, we can already handle it by processing the modalities using rewrite rules and by eliminating the existential quantifier in the resulting first-order formula using Mathematica.

The logic $\mathsf{d}\mathcal{L}$ is able to identify the natural numbers, thus our deductive system cannot be complete [Pla07b], but we have shown that it is possible to prove interesting properties of realistic systems in chapter 5. Therefore we have extended a case study [Pla07c, Pla07a] from the context of the European Train Control System [ERT02, DMO+07, DHO06]. The development of this case study has furthermore shown that large parts of the proof construction which had to be performed manually in [Pla07c] can be performed fully automatically in our theorem prover implementation.

Future Work

It has to be discussed if an extension of KeY may be feasible to check the results of Mathematica similar to the integration of Maple and PVS described in section 5.3.2.

Furthermore it may be interesting to implement calculus rules for handling existential quantifiers on the right side of the sequent and universal quantifiers on the left side beyond the universal fragment. There are two challenges for the implementation of these rules. First, in KeY, a proof is a tree and not a general directed acyclic graph. Therefore the rules that create the input for the quantifier elimination would have to close the branches that are reintegrated, when using the *simultaneous branch closing approach*. Additionally, there is no support for side deductions in KeY up to now, consequently greater changes would be necessary to implement this approach as well. Second, the check whether the rule for reintroduction of the quantifier is applicable is very expensive, as we have to check properties of all formulas in all branches below the rule application that has dropped the quantifier. It is necessary to find an implementation for this check that can be performed quickly, because every time the user clicks on a prior quantified symbol this check is performed.

As hybrid systems often consists of parallel components calculus rules should be integrated that could handle parallel systems explicitly. Currently parallelism has to be expressed either using the non-deterministic choice or the parallel composition of the components has to be precomputed by hand.

Another extension that is worth discussion is the integration of the calculus presented in [Pla07a]. It features extended modalities providing e.g. the possibility to specify that throughout a continuous evolution certain properties hold which can be useful for analysing safety at intermediate states.

Appendices

Appendix A

Design Patterns

In this section we describe the patterns [GHJV95] we use in our design.

A.1 Architectural Patterns

Architectural patterns are employed to describe the interaction between system components. For the Mathematica integration we apply the *Client Server Pattern*.

The *Client Server Pattern* is used to share responsibilities between different computes using a network. It defines two types of components. One is the server which waits for request from clients. The other is the client which is in most cases guided by an user. The client requests services from the server. The server processes the requests and replies. A common example for a client server architecture is the web browser/web server architecture used on the internet.

A.2 Fundamental Patterns

Fundamental Patterns are patterns that describe basic object oriented programming paradigms. In our context *Delegation* is used in the Mathematica interface to keep code readable. *Immutable Object* is used in the `Term` data structures, for easily comparing the data structures. To keep the code readable *Marker Interfaces* are also used.

A.2.1 Delegation

The *Delegation Pattern* enables one object to delegate requests to other objects. The requesting object does not know that the request is delegated.

A.2.2 Immutable Object

An *Immutable Object* is an object which state is set on construction time and cannot be changed. Equality checks can be performed more efficient on immutable objects. This speedup goes at cost of alternation time of this object, as it needs to be recreated if the data should be changed. A common example for an *Immutable Object* is the JAVA `String` class.

A.2.3 Marker Interface Pattern

Marker Interfaces are used to identify groups of objects that have common methods. An (marker) interface defines method stubs. A class that implements the interface has to provide the methods specified in the interface. On class interaction it can be tested whether a class implements a certain interface, thus we know that it provides certain methods.

There are also special marker interfaces, called tagging interfaces. Tagging interfaces are interfaces without method stubs. These interfaces are used to utilize polymorphism e.g. to get a common return type for a method.

The marker interface pattern is an integral part of the JAVA programming language.

A.3 Creational Patterns

Creational Patterns are patterns that deal with the creation of objects. We utilize three well known pattern in our design. First there is *Abstract Factory* which is useful for creating data structures. *Singleton* is used to restrict the number of instances of a class to one. *Lazy Initialization* is often used in combination with the singleton pattern.

A.3.1 Abstract Factory

The *Abstract Factory Pattern* is a design pattern used to make different implementations of data structures possible. An abstract factory is used as interface for creation of the data structures. There is also an interface structure that describes the data structures. The concrete implementation of the data structures is only referenced by the concrete factory, which is responsible for object creation. This way the implementation can be replaced by only using another concrete factory. An abstract representation of the pattern is shown in figure A.1.

A.3.2 Lazy Initialization

Lazy Initialization delays the creation of an object until it is needed. This is useful if an object is not always needed and the creation on run time does not brake time constraints.

A.3.3 Singleton

The *Singleton Pattern* is used when there should be only one instance of a class. This is for example useful for a class managing the available arithmetic solvers as two instances would supply exactly the same information but would cost more memory. The *Singleton Pattern* is often combined with *Lazy Initialization* since otherwise the object is created on system start up by the class loader and it is not necessarily used. Figure A.2 illustrates the combination of singleton and lazy initialization.

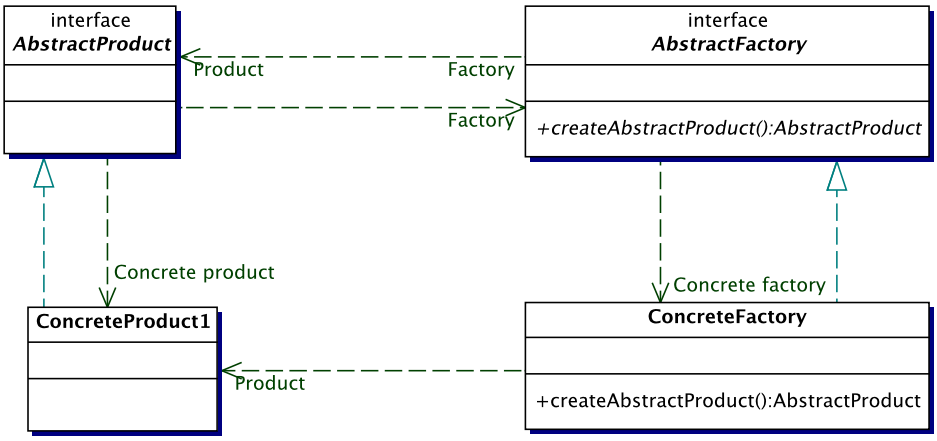


Figure A.1: Abstract class diagram for the abstract factory pattern

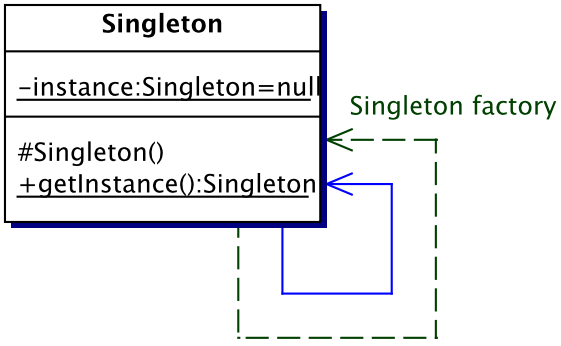


Figure A.2: Abstract class diagram for the singleton pattern with lazy initialization

A.4 Behavioral Pattern

Behavioral Patterns are abstract realizations of common problems occurring on object interaction.

A.4.1 Visitor

The visitor pattern is a useful pattern for separating algorithms from data structures. The elements of the data structure all implement a common interface that provides an *accept* method. The concrete implementation of this method calls a special method of a visitor interface that handles the concrete object. The visitor can iterate over the data structure and handle objects differently without using the reflection API to get informations about the object type. The visitor pattern is illustrated in figure A.3. An example for the use of the visitor pattern is the `PrettyPrinter` which is responsible for the output of formulas. The indention level can be considered the state of the `PrettyPrinter`. This is a huge advantage over just calling polymorphic methods [Wik07]. They would have to carry the current indention level from one method to another and would have to rely on the method implementation for handling the parameter correctly.

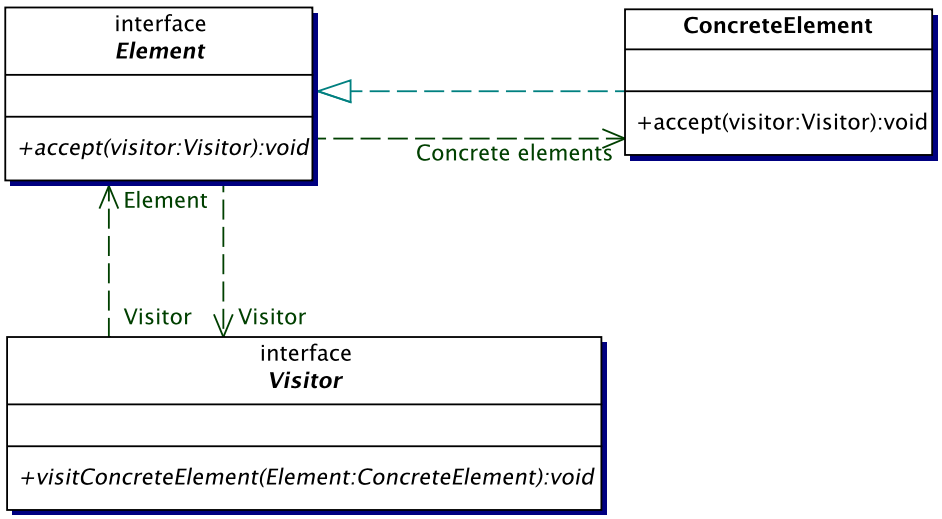


Figure A.3: Abstract class diagram for the visitor pattern

A.4.2 Iterator Pattern

The *Iterator Pattern* describes how data structures can be sequentially accessed without concrete knowledge about the structure itself. Therefore a so called iterator is provided together with the data structures that can be used it iterate over the data.

The iterator pattern is supported by JAVA and used in the JAVA class library. In JAVA 1.5 a `foreach` loop is introduced that utilizes the iterator pattern and makes code easier to write and more readable.

Bibliography

[ABB+05] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[ACHH92] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, pages 209–229, 1992.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[ADG+01] Andrew Adams, Martin Dunstan, Hanne Gottliebsen, Tom Kelsey, Ursula Martin, and Sam Owre. Computer algebra meets automated theorem proving: Integrating maple and pvs. In *TPHOLs '01: Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics*, pages 27–42, London, UK, 2001. Springer-Verlag.

[ÁMSH01a] Erika Ábrahám-Mumm, Martin Steffen, and Ulrich Hannemann. Assertion-Based Analysis of Hybrid Systems with PVS. In *Computer Aided Systems Theory - EUROCAST 2001-Revised Papers*, pages 94–109, London, UK, 2001. Springer-Verlag.

[ÁMSH01b] Erika Ábrahám-Mumm, Martin Steffen, and Ulrich Hannemann. Verification of Hybrid Systems: Formalization and Proof Rules in PVS. In *ICECCS*, pages 48–57. IEEE Computer Society, 2001.

[BBC+96] Nikolaj Bjørner, Anca Browne, Edward Y. Chang, Michael Colón, Arjun Kapur, Zohar Manna, Henny Sipma, and Tomás E. Uribe. Step: Deductive-algorithmic verification of reactive and real-time systems. In Rajeev Alur and Thomas A. Henzinger, editors, *CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 415–418. Springer, 1996.

[BBM97] Nikolaj Bjørner, Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theor. Comput. Sci.*, 173(1):49–87, 1997.

[Bec01] Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, volume 2041, pages 6–24. Springer-Verlag, 2001.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[BJK+97] Bruno Buchberger, Tudor Jebelean, Franz Kriftner, Mircea Marin, Elena Tomuta, and Daniela Vasaru. A survey of the theorema project. In *ISSAC*, pages 384–391, 1997.

[BPSM97] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible markup language (xml). *World Wide Web Journal*, 2(4):27–66, 1997.

[BSC06] Borland Software Corporation. Borland Together. `http:/www.borland.com/together/`, 2006. Last visited 12-15-2006.

[CGG83] Bruce Char, Keith Geddes, and Gaston Gonnet. The Maple symbolic computation system. *SIGSAM Bulletin (ACM Special Interest Group on Symbolic and Algebraic Manipulation)*, 17(3–4):31–42, August/November 1983.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[Che00] Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Java Series. Addison-Wesley, June 2000.

[CKOS03] E. Clarke, M. Kohlhase, J. Ouaknine, and K. Sutner. System description: Analytica 2. `http://www.cs.cmu.edu/~emc/papers/Conference%20Papers/1System%20description_Analytica%202.pdf`, Last visited 04-09-2007, 2003.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM Press, 1971.

[CZ92] Edmund M. Clarke and Xudong Zhao. Analytica - a theorem prover in mathematica. In *CADE-11: Proceedings of the 11th International Conference on Automated Deduction*, pages 761–765, London, UK, 1992. Springer-Verlag.

[DHO06] W. Damm, H. Hungar, and E.-R. Olderog. Verification of cooperating travel agents. *International Journal of Control*, 79(5):395–421, May 2006.

[DMO+07] Werner Damm, Alfred Mikschl, Jens Oehlerking, Ernst-Rüdiger Olderog, Jun Pang, André Platzer, and Boris Wirtz. Automating verification of automating verification of applications. Unpublished data, 2007.

[DS97] Andreas Dolzmann and Thomas Sturn. Redlog: computer algebra meets computer logic. *SIGSAM Bull.*, 31(2):2–9, 1997.

[EFI05] Eclipse Foundation Inc. Homepage der Eclipse-Community, November 05, 2005. `http://www.eclipse.org`.

[ERT02] ERTMS User Group, UNISIG. ERTMS/ETCS System requirements specification. `http://www.aeif.org/ccm/default.asp`, 2002. Version 2.2.2.

[Fit96] Melvin Fitting. *First-Order Logic and Automated Theorem Proving.* Springer-Verlag, New York, second edition, 1996.

[Flo67] R. W. Floyd. Assigning meanings to programs. *Proceedings Symposium on Applied Mathematics*, 19:19–31, 1967.

[Fre05] Goran Frehse. PHAVer: Algorithmic Verification of Hybrid Systems Past HyTech. In Manfred Morari and Lothar Thiele, editors, *HSCC*, volume 3414 of *Lecture Notes in Computer Science*, pages 258–273. Springer, 2005.

[Gal87] Jean H. Gallier. *Logic for Computer Science. Foundations of Automatic Theorem Proving.* John Willey and Sons, Inc., 1987.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen II. *Mathematische Zeitschrift*, 39, 1935.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley Professional, January 1995.

[GHT07] Tobias Gutzmann, Dirk Heuzeroth, and Mircea Trifu. RECODER hompage, March 14, 2007. `http://recoder.sourceforge.net`.

[Got00] Hanne Gottliebsen. Transcendental functions and continuity checking in PVS. In Mark Aargaard and John Harrison, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 197–214, Portland, OR, August 2000. Springer-Verlag.

[Hen96] Thomas Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96)*, pages 278–292, New Brunswick, New Jersey, 1996.

[HHWT97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH: A Model Checker for Hybrid Systems. *International Journal on Software Tools for Technology Transfer*, 1(1–2):110–122, 1997.

[HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic logic.* MIT Press, 2000.

[Hui01] Marieke Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle.* PhD thesis, University of Nijmegen, The Netherlands, 2001.

[HWL+02] Marc Hoy, Dave Wood, Marc Loy, James Elliot, and Robert Eckstein. *Java Swing*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[Int83] International Organization for Standardization. *ISO Standard 646, 7-Bit Coded Character Set for Information Processing Interchange*. International Organization for Standardization, Geneva, Switzerland, second edition, 1983. Also available as ECMA-6.

[JKP04] T. Jebelean, L. Kovacs, and N. Popov. Experimental Program Verification in the Theorema System. In T. Margaria and B. Steffen, editors, *Proceedings ISOLA 2004*, pages 92–99, Paphos, Cyprus, November 2004.

[JP04] Jörn Pachl. *Systemtechnik des Schienverkehrs – Bahnbetrieb planen, steuern und sichern*. Teubner, Stuttgart, Germany, 2004.

[KeY07] The KeY Project – Integrated Deductive Software Design. `http://www.key-project.org/`, 2007. Last visited 04-10-2007.

[LPC+07] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R. Cok, Peter Müller, Joseph Kiniry, and Patrice Chalin. JML reference manual. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`, February 2007.

[MS98] Zohar Manna and Henny Sipma. Deductive verification of hybrid systems using step. In *HSCC '98: Proceedings of the First International Workshop on Hybrid Systems*, pages 305–318, London, UK, 1998. Springer-Verlag.

[Old02] Ernst-Rüdiger Olderog. Logik – Vorlesungsskript zum Modul Theoretische Informatik I. Vorlesungsskript, Universität Oldenburg, 2002. In German. Available at: `http://csd.informatik.uni-oldenburg.de/~skript/Logik/SS02/Skript/logik.pdf`.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[Pau94] Lawrence C. Paulson. Isabelle: a generic theorem prover. In *Lecture Notes in Computer Science*, volume 828, New York, NY, USA, 1994.

[Pla04] André Platzer. An object-oriented dynamic logic with updates. Master's thesis, University of Karlsruhe, Department of Computer Science. Institute for Logic, Complexity and Deduction Systems, September 2004.

[Pla07a] A. Platzer. A temporal dynamic logic for verifying hybrid system invariants. In S. Artemov and A. Nerode, editors, *Logical Foundations of Computer Science, International Symposium, LFCS 2007, New York, USA, Proceedings*, volume 4514 of *LNCS*, pages 457–471. Springer,

2007. http://www.springer.com/comp/lncs/index.html(c) Springer-Verlag.

[Pla07b] A. Platzer. Towards a hybrid dynamic logic for hybrid dynamic systems. In Patrick Blackburn, Thomas Bolander, Torben Braüner, Valeria de Paiva, and Jørgen Villadsen, editors, *Proc., LICS International Workshop on Hybrid Logic, HyLo 2006, Seattle, USA*, ENTCS, 2007. To appear at http://www.elsevier.nl/locate/entcs/ENTCS.

[Pla07c] André Platzer. Differential dynamic logic for verifying parametric hybrid systems. In Nicola Olivetti, editor, *TABLEAUX*, LNCS. Springer, 2007.

[PQ95] Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25(7):789–810, 1995.

[SSM04] Sriram Sankaranarayanan, Henny Sipma, and Zohar Manna. Constructing invariants for hybrid systems. In Rajeev Alur and George J. Pappas, editors, *HSCC*, volume 2993 of *Lecture Notes in Computer Science*, pages 539–554. Springer, 2004.

[Sun05] Sun Microsystems, Inc. Swing (Java Foundation Classes). http://java.sun.com/javase/6/docs/technotes/guides/swing/index.html, 2005. Last visited 03-21-2007.

[Tar51] A. Tarski. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, 2d edition, 1951.

[Wik07] Wikimedia Foundation, Inc. Visitor pattern – Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Visitor_pattern, 2007. Last visited 03-21-2007.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Wol03] Stephen Wolfram. *The Mathematica Book*. Wolfram Media, Incorporated, 2003.

[Wol07] Wolfram Research, Inc. Java Toolkit: J/Link: Integrating Mathematica and Java, March 08, 2007. http://www.wolfram.com/solutions/mathlink/jlink.

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmitteln und Quellen benutzt habe.

Ort	Datum	Unterschrift