Proof Support for Duration Calculus

Søren T. Heilmann

January 3, 1999

This dissertation is submitted to the Department of Information Technology of the Technical University of Denmark in partial fulfillment of the requirements for the degree of Doctor of Philosophy. The work has been supervised by Associate Professor Anders P. Ravn and Associate Professor Hans Rischel.

Lyngby, January 1999

Søren T. Heilmann

Bevisunderstøttelse af Varighedskalkylen

Resumé

Vi udvikler bevisføreren Isabelle/DC til real-tids logikken Duration Calculus (Varighedskalkyle). Udgangspunktet er en definition af klassisk Duration Calculus logic i formuleret i klassisk Hilbert stil. Vi udvikler en sekvent logik formularing af DC og dokumenterer dens relation til Hilbert stil versionen.

Med henblik på anvendelserne generaliserer vi klassisk DC og følger Ravn når vi introducerer typede tilstandsvariable og de ekstra operatorer i Extended DC til at udtrykke egenskaber ved kontinuerte tilstande.

PC/DC er en eksisterende bevisfører for DC udviklet af Skakkebæk og baseret på PVS, en bevisfører for højere ordens logik. Vi har brugt PC/DC til verifikation af Steam Boiler eksemplet af Rischel oa. Vi dokumenterer erfaringerne fra dette arbejde og konkluderer at der er et behov for mere kraftfulde beviskommandoer end dem, der findes i PC/DC. Ydermere har vores arbejde med at vedligeholde og videreudvikle PC/DC fået os til at se os om efter en alternativ platform for en DC bevisfører.

Vi vælger Isabelle som vores alternative platform og dokumenterer Isabelles basalle logik samt dets faciliteter for udvidelse af denne logik. Dette danner grundlag for den efterfølgende præsentation af indkodningen af den tidligere udviklede DC sekvent logik.

Vi dokumenterer vores arbejde med at udvikle bevisunderstøttelse for DC i det nye værktøj. Dette består i en samling af afledte regler og lemmaer som har vist sig brugbare. Vi bruger også bevisprogrammeringsfaciliteterne i Isabelle og dokumenterer de taktikker (tactics) og taktikaler (tacticals) vi har udviklet til bevissøgning og højniveau bevisskridt.

Vi vender tilbage til Steam Boiler eksemplet. Denne gang udstyret med et bevisværktøj som vi let kan udvide med bevisunderstøttelse for det valgte eksempel og forhåbenligt også generelt. Vi dokumenterer indkodningen af eksemplet og de taktikker vi udvikler. Da vi nu har to versioner af det samme bevis skabt med to forskellige værktøjer kan vi sammenligne disse beviser og konkluderer at Isabelle/DC tilbyder forbedret bevisunderstøttelse.

Abstract

We develop a theorem prover, Isabelle/DC, for the real-time logic Duration Calculus based on the generic proof tool Isabelle. Our starting point is a definition of classical Duration Calculus logic in the Hilbert style. We develop a sequent style version of DC and document its properties in relation to the Hilbert style version.

With a view towards applications we generalize classical DC and introduce typed states in the style of Ravn and the extra operators of Extended DC for expressing properties of continuous states.

PC/DC is an existing theorem prover for DC developed by Skakkebæk and based on PVS, a proof tool for higher order logic. We have worked with PC/DC to verify a specification of the Steam Boiler by Rischel et al. We document the experiences of this work and conclude that there is a need for more powerful proof commands than found in PC/DC. Furthermore our work on supporting and further developing PC/DC has lead us to look for a alternative platform for a DC theorem prover.

Isabelle is the alternative platform we choose for implementation of our proof tool. We document the basic logic of Isabelle and the features it provides for extension. This gives us a basis for presenting the encoding of the sequent style DC developed previously.

We document our work on providing proof support for DC in the new tool. This is a collection of derived rules and lemmas which have proved themselves useful. We also make use of the proof programming facilities of Isabelle and document tactics and tacticals we have developed for proof search and high level proof steps.

We revisit the Steam Boiler case study. This time equipped with a proof tool that we can extend easily to provide proof support suitable for this example and hopefully also generally. We document the encoding of the Boiler specification and the tactics we develop. Having two version of the same proofs done with different tools we compare them and find that Isabelle/DC provide improved proof support over PC/DC.

Acknowledgements

First of all I would like thank my supervisor Anders P. Ravn for the guidance he has provided, for the fruitful discussions and for his patience. Thanks also goes to my co-supervisor Hans Rischel. He was always open for discussions and assistance when the need arose.

I will also thank Professor Ernst-Rüdiger Olderog for providing me with an opportunity for visiting his department in Oldenburg. During the stay I was fortunate to work with Martin Fränzle and Henning Dierks. I thank them both for their help and assistance and also Martin for his kind hospitality.

Both staff and students at the department in Lyngby have contributed to the enjoyable stay I had there for three years. Special thanks goes to Michael Reichhardt Hansen for his consultancy on Duration Calculus, to Jens Ulrik Skakkebæk for getting me started with PVS and to Morten Ulrik Sørensen for being my LaTeX guru. Ken Larsen and Jakob Lichtenberg provided invaluable assistance with all things Linux.

Last but not least I would like to thank my good friend Jakob Frost for sharing his knowledge of Isabelle and for his moral support. And also Simon Mørk, with whom I shared office, for fruitful discussions, many a good laugh and for sharing the joy of being a PhD student.

Contents

Chapter 1

Introduction

Did you ever have car trouble? I did. At the time I started my PhD-studies I bought my first car. It was a new car of italian origin, a new model even, and I paid the equivalent of 10 months pay for it. If you have ever owned a new car you know that it is something quite special. My car was special indeed, it would not run.

Every once in a while when driving the engine would simply stop for no apparent reason. Apart from being a nuisance this feature also brought about some rather hazardous situations. Approaching a bend in heavy traffic doing 80 km/h and having the engine and power steering suddenly die on you is no fun. Not even compared to the difficulty you have convincing the dealer that something is wrong.

Eventually the problem was fixed. It turned out to be a software glitch in the engine controller and I came away from the ordeal with a working car and the personal experience that some computer applications really are safety critical.

Erroneous software is not particular to engine controllers of course, but exists everywhere computers are in control. *Embedded real-time systems* are systems where a computerized controller interacts with an environment as it evolves over time. The engine controller in my car is such a system and my little story demonstrates that such systems do not always work correctly and furthermore it shows that safety may be jeopardized when they do not.

1.1 Proof Based Systems Engineering

Formal methods has been suggested as a way of improving the quality of software in general. The focus is on using precise mathematically well-founded models, notations and techniques for the specification of system requirements and design. Verification, a process of rigorously applying rules of the formal method, is used to ensure that the design has the properties expressed by the requirements.

The Duration Calculus (DC) [9] is a real-time logic proposed to capture and reason about both requirements and designs of embedded real-time systems. Verification is done through calculation using the rules of the logic.

In the Duration Calculus a system is modeled as a collection of state variables interpreted as boolean valued functions of time. Time is continuous and modeled by the real numbers. A DC formula describes system behavior by describing the collection of valid interpretations of the state variables over a time interval.

To illustrate the approach of proof based development we return for a moment to the engine controller of a car and consider a hypothetical overload protection subsystem of that controller. The engine overheats if it runs at too many revolutions per minute (rpm) for too long and it is the job of the subsystem to prevent this.

We identify one state variable *Overload* for the system:

$$Overload \quad \in \quad Time \rightarrow \{0, 1\}$$

The reader can think of *Overload* as connected to the rpm meter on the dashboard of the car. When the meter is in the red area of the scale there is an overload.

The engine should be allowed to be overloaded for short periods of time for example during acceleration but should not be driven at high rpms for long periods as this causes build up of heat in the engine block and increased internal friction.

To avoid this heat build up there is a requirement that in any time interval longer than 60 seconds the engine overloads in only 1 tenth of that time.

If the time interval $[b, e]$ is from b to e with $e \geq b$ we can measure the duration that the engine overloads by the integral of *Overload*

$$\int_b^e Overload(t)\,dt$$

and use this measure to express the overload requirement in normal mathematical notation:

$$e - b > 60 \Rightarrow 10 \int_b^e Overload(t)\,dt \leq (e - b)$$

In the Duration Calculus time and the interval over which we consider the system are implicit. We can therefore formulate the requirement as:

$$Req \quad \hat{=} \quad \ell > 60 \Rightarrow 10 \textstyle\int Overload \leq \ell$$

where ℓ denotes the interval length and may be defined as an abbreviation of $\int 1$.

We now turn to the design of the system and states two high level design decisions for the protection system. The first decision is to detect and limit the duration of all overloads to 3 second. We express this in the Des_1 formula:

$$Des_1 \quad \hat{=} \quad \Box(\lceil Overload \rceil \Rightarrow \ell \leq 3)$$

where $\lceil Overload \rceil$ abbreviates $\int Overload = \ell \land \ell > 0$ and $\Box A$ means that A holds in all subinterval.

Secondly we decide to interleave any two overloads with a 30 second period of non-overload. Note that we can enforce these design decisions because we can stop any overload by simply turning of the fuel supply to the engine. We formulate the second design decision as:

$$Des_2 \;\hat{=}\; \Box(\lceil Overload \rceil \frown \lceil \neg Overload \rceil \frown \lceil Overload \rceil \Rightarrow \ell \geq 30)$$

where $\frown$ is the binary operator for chopping an interval into two consecutive subintervals. The formula states that if the interval under consideration can be described as consecutive sub-intervals of overload, no overload and then overload again then the interval will be longer than 30 second.

The design process will continue to break down these design decisions further but already at this stage we can check that we are on the right track. We do this by verifying that the design satisfies the requirements. This is expressed by implication in the logic. The judgement we want to prove is:

$$Protect \;\hat{=}\; Des_1 \land Des_2 \Rightarrow Req$$

An informal hand proof of this formulas takes up a few pages of arguments. We will no present the entire proof here but illustrate the process with a proof of the following lemma Lem:

$$T \;\hat{=}\; 10 \int Overload \leq \ell$$

$$Lem \;\hat{=}\; \begin{array}{l} Des_1 \land Des_2 \land (T \frown \lceil Overload \rceil \frown \lceil \neg Overload \rceil \frown \lceil Overload \rceil) \Rightarrow \\ \quad T \frown \lceil Overload \rceil \end{array}$$

We present the proof as a chain of implications:

$$\begin{array}{lll} Des_1 \land Des_2 \land (T \frown \lceil Overload \rceil \frown \lceil \neg Overload \rceil \frown \lceil Overload \rceil) & \Rightarrow & \{Des_1, Des_2\} \\ T \frown (\lceil Overload \rceil \land \ell \leq 3) \frown (\lceil \neg Overload \rceil \land \ell \geq 30) \frown \lceil Overload \rceil & \Rightarrow & \{Thm_{1,2,3}\} \\ T \frown (\int Overload \leq 3 \land \ell \geq 30) \frown \lceil Overload \rceil & \Rightarrow & \{\text{arithmetic}\} \\ T \frown T \frown \lceil Overload \rceil & \Rightarrow & \{Thm_3\} \\ T \frown \lceil Overload \rceil & & \end{array}$$

In the proof we appeal to the following three general lemmas of DC:

$$\begin{array}{lll} Thm_1 & \hat{=} & \int P + \int \neg P = \ell \\ Thm_2 & \hat{=} & \lceil P \rceil \Rightarrow (\int \neg P = 0) \\ Thm_3 & \hat{=} & (r_1 \leq \int P \leq u_1) \frown (r_2 \leq \int P \leq u_2) \Rightarrow (r_1 + r_2 \leq \int P \leq u_1 + u_2) \end{array}$$

We also implicitly use two basic rules of DC, the monotonicity and associativity of chop:

$$
\begin{array}{lll}
\text{M1}: & \text{if } A \Rightarrow B \text{ then } A \frown C \Rightarrow B \frown C & \text{Monotonicity} \\
\text{M2}: & \text{if } A \Rightarrow B \text{ then } C \frown A \Rightarrow C \frown B & \\
\text{A2}: & A \frown (B \frown C) \Leftrightarrow (A \frown B) \frown C & \text{Associativity}
\end{array}
$$

1.2 Mechanical Verification

For anything but the smallest examples doing specification and verification using pen and paper only is not feasible. Tool support is necessary to ensure that specifications conform to the models and notations of the chosen formalism and that every step of a verification applies the rules of reasoning correctly.

It is instructive to see how a specification like the overload protection system is captured in a proof tool. For this we use the proof tool PC/DC developed by Skakkebæk [45]. The specification appears in figure 1.1. Initially we have the declaration of *Overload* as a boolean state and the declaration of the formulas Reg, Des_1, Des_2 and T. At the end we state two lemmas `Line2` and `Lem` and the main theorem `Protect`. `Line2` captures the reasoning in line 2 of the hand proof above as a separate lemma. Although the notation for this specification is in ASCII it is still quite close to the one used above.

This however is not true for the mechanized proofs of `Lem` and `Line2` which appear in figures 1.2 and 1.3, respectively. We show these proofs merely to give an impression of their size. Each node in the proof trees is a proof command issued by the user of the tool during the interactive proof session.

What above appears to be a linear 4 step argument reveals itself as a branched proof consisting of some 60 proof steps. If we zoom in on a part of the proof of `Line2` we see some of the reason for this blow up. Consider figure 1.4. At this point in the proof the proof obligation is reduced to:

$$
\begin{array}{l}
(r_2 \leq r_1 \land r_2 \leq 3) \frown (r_3 \geq 30) \land \\
(\int tt = r_1) \frown (\int tt = r_3) \land \\
(\int(Overload) = r_2) \frown (\int(Overload) = 0) \Rightarrow \\
\int(Overload) \leq 3 \land \ell \geq 30
\end{array}
$$

which is in the sequent style logic of PC/DC looks like below. The assumptions appear as numbered formulas above the sequent line and the conclusion appear below the line. PC/DC use `;` as the operator symbol for chop. `tt` is the always true state so `len = dur(tt)`.

```
overload : DCTHEORY
  BEGIN

    Overload : State[bool]

    Req : IForm = len >= n(60) => n(10)*dur(Overload) <= len;;

    Des1 : IForm = []('Overload' => len <= n(3));;
    Des2 : IForm =
       []('Overload';'NOT Overload';'Overload' => len>=n(30));;

    T : IForm = n(10)*dur(Overload) <= len;;

    Line2 : LEMMA (len <= n(3));('NOT Overload' /\ len >= n(30)) =>
                   (dur(Overload) <= n(3) /\ len >= n(30))

    Lem : LEMMA Des1 /\ Des2 /\ T;'Overload';'NOT Overload';'Overload' =>
               T;'Overload'

    Protect : THEOREM Des1 /\ Des2 => Req

  END overload
```

Figure 1.1: PC/DC specification of Overload Protection design

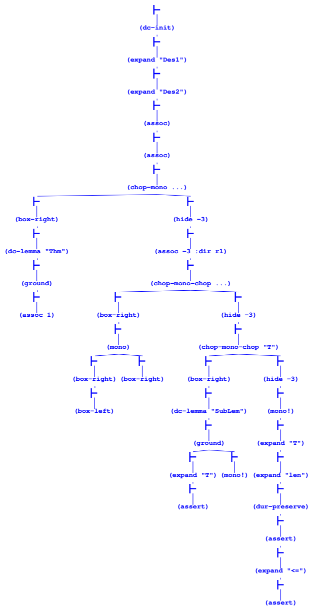


Figure 1.2: PC/DC proof of `Lem`

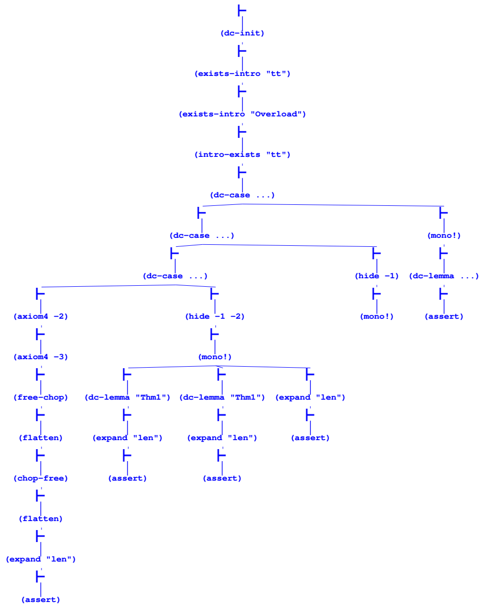


Figure 1.3: PC/DC proof of `Line2`

```
{-1}    (r!2 <= r!1 /\ r!2 <= 3);(r!3 >= 30)
[-2]    (dur(tt) = r!1);(dur(tt) = r!3)
[-3]    (dur(Overload) = r!2);(dur(Overload) = 0)
  |-------
[1]    dur(Overload) <= 3 /\ len >= 30
```

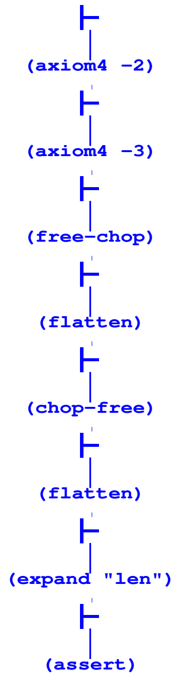


Figure 1.4: Excerpt of PC/DC proof of line 2

The following is an commented (and somewhat edited) transcript of the proof session for this small part of the proof. To aid readability formulas in a sequent which change in a proof step is marked by numbers in curly brackets (eg. {-1}).

```
{-1}    (r!2 <= r!1 /\ r!2 <= 3);(r!3 >= 30)
[-2]    (dur(tt) = r!1);(dur(tt) = r!3)
[-3]    (dur(Overload) = r!2);(dur(Overload) = 0)
  |-------
[1]    dur(Overload) <= 3 /\ len >= 30
```

Durations sum up over chop so the duration of `tt` on the whole interval is `r!1+r!3`. We issue the command `(AXIOM4 -2)` to exploit this and get:

```
[-1]    (r!2 <= r!1 /\ r!2 <= 3);(r!3 >= 30)
{-2}    dur(tt) = r!1 + r!3
[-3]    (dur(Overload) = r!2);(dur(Overload) = 0)
  |-------
[1]    dur(Overload) <= 3 /\ len >= 30
```

Then we do the same for the `Overload` state. Using the command `(AXIOM4 -3)` we get:

```
[-1]    (r!2 <= r!1 /\ r!2 <= 3);(r!3 >= 30)
[-2]    dur(tt) = r!1 + r!3
{-3}    dur(Overload) = r!2
  |-------
[1]    dur(Overload) <= 3 /\ len >= 30
```

From the relations on the constants in `[-1]` and arithmetic reasoning it is not difficult to see that the sequent is true and an informal proof would end here. However the mechanized proof still has to go through several explicit steps. First we exploit that the relation `r!2 <= r!1 /\ r!2 <= 3` is context free ie. independent of the interval. This means that it can be distributed over the chop to the top level. It is necessary to bring this information to the top level in order for the arithmetic reasoning to exploit it. We issue the command `(FREE-CHOP)` and get:

```
{-1}    (r!2 <= r!1 /\ r!2 <= 3) /\ (TRUE;(r!3 >= 30))
[-2]    dur(tt) = r!1 + r!3
[-3]    dur(Overload) = r!2
  |-------
[1]    dur(Overload) <= 3 /\ len >= 30
```

We now have to explicitly rearrange the sequent using the `(FLATTEN)` command and get:

```
{-1}    r!2 <= r!1
{-2}    r!2 <= 3
{-3}    TRUE;(r!3 >= 30)
[-4]    dur(tt) = r!1 + r!3
[-5]    dur(Overload) = r!2
  |-------
[1]    dur(Overload) <= 3 /\ len >= 30
```

The relation `r!3 >= 30` is also context free. We distribute it to the top level using the command `(CHOP-FREE)` and get:

```
[-1]    r!2 <= r!1
[-2]    r!2 <= 3
{-3}    r!3 >= 30 /\ (TRUE;TRUE)
[-4]    dur(tt) = r!1 + r!3
[-5]    dur(Overload) = r!2
  |-------
[1]    dur(Overload) <= 3 /\ len >= 30
```

Again we (FLATTEN) and get:

```
[-1]    r!2 <= r!1
[-2]    r!2 <= 3
{-3}    r!3 >= 30
{-4}    TRUE;TRUE
[-5]    dur(tt) = r!1 + r!3
[-6]    dur(Overload) = r!2
  |-------
[1]    dur(Overload) <= 3 /\ len >= 30
```

Then we explicitly have to exploit that `len=dur(tt)`. We expand this definition using the command (EXPAND "len") and get:

```
[-1]    r!2 <= r!1
[-2]    r!2 <= 3
[-3]    r!3 >= 30
[-4]    TRUE;TRUE
[-5]    dur(tt) = r!1 + r!3
[-6]    dur(Overload) = r!2
  |-------
{1}    dur(Overload) <= 3 /\ (dur(tt) >= 30)
```

Finally we can invoke the arithmetic decision procedure (ASSERT) to prove the goal:

```
This completes the proof
```

and ends our illustration of the details involved in mechanized reasoning. The aim of our work is to reduce this detail and we do this by constructing a new proof tool for Duration Calculus called Isabelle/DC.

1.3 Related work

There is a wast literature on logic, mechanized reasoning etc. In this subsection we focus on the work relevant for mechanizing Duration Calculus.

1.3.1 DC Proof Assistants

The primary related work is the PC/DC proof assistant by Skakkebæk [45]. Our work with PC/DC and the experiences gained by extending it and using it for case studies is what prompted us to take a different approach to providing proof support for DC. We reserve Chapter 3 for an in depth description of PC/DC and its implementation.

Work by Haxthausen and Yong [24] encodes DC semantically in the specification language RSL to develop the justification assistant DC/RJ^{-}. RSL provides a rich specification language with higher order types, CCS like processes and imperative features. RSL is supported by the tool set RAISE including a proof assistant. The proof assistant has a powerful simplifier but otherwise has no tactical support.

PC/DC and DC/RJ^{-} mainly differ in the way proofs are performed. Proofs in PC/DC expand the meaning of the DC logical operators. This reveal the underlying logical operators of the PVS logic and allow reasoning on these using the rich set of proof commands of PVS. Proofs in DC/RJ^{-} does not expand the DC operator definitions. Instead a number of rules for propositional logic reasoning is encoded for the DC operators.

Mao el al. [50] have developed the proof assistant DC/P^{-} for a family of interval logics based on Neighborhood Logic a logic containing DC. The encoding is semantic and done in PVS. It differs from the ones in PC/DC and DC/RJ^{-} in the treatment of global and temporal variables of NL. In DC/P^{-} they are explicitly represented as a type and interpretation and valuation functions map variables to their meaning. This is a very traditional way of giving semantics to formulas. In PC/DC and DC/RJ^{-} variables are directly identified with PVS variables representing their meaning. This way the type systems of PVS and RSL are extended to the encoded DC logic.

1.3.2 Isabelle

We base our tool Isabelle/DC on the generic proof tool Isabelle by Paulson [33]. Isabelle has many other uses. The distribution of Isabelle comes with encodings of First Order Logic, Higher Order Logic and Zermelo-Frankel set theory. Many useful tricks on the use of Isabelle can be gleaned from this work. These encodings are however different from ours in that they support the proof style of Natural Deduction [38] where we use a sequent formulation [17]. Frost [16] also use a sequent formula for his encoding of OLE, Operational Logic of Effects. This thorough work contains interesting and important considerations for proving an encoding faithful to the original formulation.

1.3.3 Decision Procedure for DC

Work on decision procedure for subsets of DC is important to us because their use can save the user much detailed reasoning.

Several people have worked on the fundamental question of decidability of various subsets of DC. Zhou, Hansen and Sestoft [51] investigates decidability of a subset of DC called RDC built from state predicates ($\lceil P \rceil$) combined with chop ($\frown$) and the propositional connectives. They find that RDC is decidable for both discrete and continuous time DC. They go on to extend RDC in three other directions and defined $RDC_1(k)$, RDC_2 and RDC_3. $RDC_1(k)$ introduce $\ell = k$ where k is a constant as atomic formulas together with state predicates $\lceil P \rceil$. RDC_2 is build from atomic formulas $\int P = \int Q$ combined with chop and propositional connectives. RDC_3 is build from state predicates and formulas $l = x$ where x is a global variable connected by chop, proposition connectives and quantification. Hansen and Zhou find that $RDC_1(k)$, RDC_2 and RDC_3 are all undecidable.

Fränzle [15] extends the results of Hansen and Zhou by considering other syntactical subsets as well as semantical models. Interestingly he considers the semantical subsets of n-bounded states ($n \geq 2$). n-bounded states are states that vary at most n times on a unit interval. These are models of systems build from asynchroneously clocked controllers. He find that the subset $RDC_1(k)$ (investigated by Hansen and Zhou) is decidable on n-bounded models but RDC_2 and RDC_3 are not. This result is based on a relation of n-bounded model with timed regular expressions [5] and timed automata [4]. As a special result he finds that formulas build from state predicates and the formulas $\ell > k$, $\ell = k$ and $\ell < k$ and combined with propositional connectives except negation is decidable on the full model of finitely variable states.

Lakhnech [28] considers standard forms of DC [25, 40]. He shows that DC formulas restricted to this subset is decidable via a reduction to the emptiness problem of a class of hybrid automata for which this problem is decidable.

The results of Hansen and Zhou on the decidability of RDC has been exploited in the construction of a decision procedure for this subset by Skakkebæk and Sestoft [46]. This procedure is implemented in SML and used in the PC/DC tool.

Pandya [32] has a program called DCVALID for checking validity of discrete time Duration Calculus formulae. It builds on Mona [27], a theorem prover for second order monadic logic.

Zhou et al. [10] consider Linear Duration Invariants (LDI) as requirement specification of systems modeled by real-time automata. An LDI is a linear inequality of the form $\sum_{i=1}^{k} c_i \int S_i \leq M$. Satisfaction of an LDI by a real-time automaton degenerates to deciding a collection of linear programming problems. A tool implementing this approach is described in [13].

1.3.4 Real Arithmetic

Real arithmetic is a fragment of DC for which decision procedure support is absolutely essential for reasoning easily about duration measures. The elementary (first order) theory of reals is both complete and decidable [48] but the existing decision procedures all have terrible algorithmic complexity. Harrison [22] develops a constructive theory of the real numbers in HOL [18] and implements a quantifier elimination procedure. A more modest but realistic choice is to settle for quantifier free linear arithmetic for which several effective procedures exists. The Stanford Validity Checker (SVC), developed by Dill et al. [6], follows the approach advocated by Shostak [44] and combines the decision of several logics into one efficient procedure. SVC decides propositional logic, the theory of equality and congruence of uninterpreted functions, linear real arithmetic and theories of bit vectors, arrays and records. We use SVC in our tool Isabelle/DC described in Chapter 6.

1.4 Contribution

The contribution of this work is the development of Isabelle/DC the theorem prover for Duration Calculus based on Isabelle. We have pursued the approach of constructing a theorem prover by a syntactic encoding of the language and proof system of the logic. This has resulted in a tool supporting the user with a convenient syntax for specifications and a proof assistant with automatic verification support for a number of recurring proof patterns.

Further we have demonstrated that integrating this proof assistant with stand alone decision procedures for logical fragments can be done easily and lead to a usable and efficient result.

We demonstrate the use of Isabelle/DC on a case study and compare this work with similar work done with PC/DC.

We find that the open and well-documented nature of the underlying tool Isabelle contributed in an essential way to easing the development of this tool and we expect that these characteristics will also ease support and further development of the tool in the future.

1.5 Overview

The structure of the thesis is as follows: Chapter 2 introduces the Duration Calculus and investigates its properties with the precision necessary to form a basis for the following chapters. This chapter also contains a discussion of ways of mechanizing the logic. Chapter 3 describe the PC/DC tool developed by Skakkebæk and in Chapter 4 we verify the specification of the Steam Boiler. In this case study we use the PC/DC to do the reasoning and we gain experience using this tool. The experience gained we put into the development of the new prover for DC. We choose to build our own tool on the generic Isabelle theorem prover and devote Chapter 5 for the description of this tool. Finally Chapters 6, 7 and 8 contain descriptions of out new tool Isabelle/DC and its implementation. In Chapter 9 we revisit the Steam Boiler to apply the tool on this case study and in Chapter 10 discuss the result of the work. Additionally there are a number of appendices providing background information. Appendix F describes the accompanying CD–ROM.

Chapter 2

Logic

In this chapter we present the Duration Calculus (DC). This elaborate presentation of DC and its proof systems is to serve as a well defined basis for our further work on tool support for DC described in the following chapters.

We stress the point of introducing a sequent style proof system for DC as it is well-known that proof systems in this form is amenable for tool support which systems in the Hilbert style is not.

Our presentation will follow the one by Hansen and Zhou [21] closely. We start by defining the Interval Logic on which DC is built by presenting its syntax, semantics and Hilbert style proof system as defined in [21]. We then define a sequent style proof system for Interval Logic inspired by the one found in [45]. Having developed the logic in a new style we want to understand its relation to the classical Hilbert style formulation. We therefore investigate its properties and relation to the system in [21].

Having defined the Interval Logic we define DC as an extension. This involves extending the syntax, semantics and proof system. Again our presentation is close to the one in [21]. We also defined a DC extension to the proof system in the sequent style.

2.1 Interval Logic

2.1.1 Syntax

The formulas of IL are generated inductively from the following sets of symbols.

x, y, z	$\in$	$GVar$	Global variables
v	$\in$	$TVar$	Temporal variables
f^n, g^m	$\in$	$FSymb$	Function symbols
G^n, H^m	$\in$	$RSymb$	Relation symbols
X	$\in$	$PLetter$	Propositional letters

Function symbols f^n and relation symbols G^n are decorated with their arity n, the number of arguments they take.

The inductive definition is given by the syntax for terms and formulas:

$$
\begin{array}{lcll}
t & ::= & x \mid f^n(t_1, \ldots, t_n) \mid \ell \mid v & \text{Terms} \\
A, B, C & ::= & X \mid G^n(t_1, \ldots, t_n) \mid \neg A \mid A \vee B \mid A \frown B \mid (\exists x \cdot A) & \text{Formulas}
\end{array}
$$

The syntax is ordinary first order predicate logic with a few extension. ℓ denotes the interval length and $\frown$ is the *chop* operator enabling the interval to be chopped up and described by formulas for its left and right part.

The remaining logical connectives, quantifiers and modalities are defined by the usual abbreviations:

$$
\begin{array}{lcll}
\mathit{true} & \hat{=} & \ell \geq 0 & \\
\mathit{false} & \hat{=} & \neg \mathit{true} & \\
A \wedge B & \hat{=} & \neg(\neg A \vee \neg B) & \\
A \Rightarrow B & \hat{=} & \neg A \vee B & \\
A \Leftrightarrow B & \hat{=} & (A \Rightarrow B) \wedge (B \Rightarrow A) & \\
(\forall x \cdot A) & \hat{=} & \neg(\exists x \cdot \neg A) & \\
\Diamond A & \hat{=} & \mathit{true} \frown A \frown \mathit{true} & \text{Somewhere} \\
\Box A & \hat{=} & \neg \Diamond \neg A & \text{Always}
\end{array}
$$

To cut down on the number of brackets needed when writing formulas we assign precedence to operators and connectives. In descending order the precedence is:

$$\text{(high)} \quad \neg, \Box, \Diamond, \frown, \wedge, \vee, \Rightarrow, \Leftrightarrow \quad \text{(low)}$$

All binary operators are right associative. Quantified expressions will always be bracketed to clarify the extent of the binding.

2.1.2 Semantics

The meaning of formulas and terms will depend on the meaning assigned to the basic symbols of the language. First we introduce some basic domains.

Time and interval The model for time is the real numbers $\mathbb{R}$. Time intervals are denoted by their beginning and end-points. Intervals are closed and contain all time points in between.

$$
\begin{array}{rcl}
\mathbb{T}\text{ime} & = & \mathbb{R} \\
\mathbb{I}\text{ntv} & = & \{[b, e] \mid b, e \in \mathbb{T}\text{ime} \wedge b \leq e\}
\end{array}
$$

Function and relational symbols We will assume a fixed meaning of functional and relational symbols. The meaning of f^n is an n-ary function $\overline{f}^n$. The meaning of G^n will be an n-ary relation $\overline{G}^n$. In particular we will assume that $+, -, *, \ldots$ and $=, \leq, \ldots$ have their usual meaning.

For presentational convenience we only consider real-valued terms and functions. In this way we postpone the need to define the concepts of well-typed formulas and terms to section 2.3.

Global variables The meaning of global variables is captured by a *value assignment* $\mathcal{V}$ being simply a map from $GVar$ to the reals.

$$\mathcal{V} \in \mathbb{V}\mathrm{al} = GVar \to \mathbb{R}$$

Temporal variables and propositional letters The meaning of temporal variables and propositional letters is captured collectively by an *interpretation* $\mathcal{I}$.

$$\mathcal{I} \in \mathbb{I}\mathrm{ntp} = \begin{pmatrix} TVar \\ \cup \\ PLetter \end{pmatrix} \to \begin{pmatrix} \mathbb{I}\mathrm{ntv} \to \mathbb{R} \\ \cup \\ \mathbb{I}\mathrm{ntv} \to \mathbb{B} \end{pmatrix}$$

with the restriction that $\mathcal{I}(v) \in \mathbb{I}\mathrm{ntv} \to \mathbb{R}$ and $\mathcal{I}(X) \in \mathbb{I}\mathrm{ntv} \to \mathbb{B}$. Where $\mathbb{B}$ is the set of boolean values $\{true, false\}$.

Formulas The meaning of formulas is defined inductively over the syntax by the function $[\![\cdot]\!]$.

$$[\![A]\!] \in \mathbb{I}\mathrm{ntp} \to (\mathbb{V}\mathrm{al} \times \mathbb{I}\mathrm{ntv} \to \mathbb{B})$$

We will use $\mathcal{I}[\![A]\!](\mathcal{V}, [b, e])$ instead of the more traditional syntax of $[\![A]\!](\mathcal{I})(\mathcal{V}, [b, e])$ for application and write $\mathcal{I}, \mathcal{V}, [b, e] \models A$ iff $[\![A]\!](\mathcal{I})(\mathcal{V}, [b, e])$ is true.

$$\begin{array}{lll}
\mathcal{I}, \mathcal{V}, [b, e] \models X & \text{iff} & \mathcal{I}(X) \\
\mathcal{I}, \mathcal{V}, [b, e] \models G^n(t_1, \ldots, t_n) & \text{iff} & \overline{G}^n(\mathcal{I}[\![t_1]\!](\mathcal{V}, [b, e]), \ldots, \mathcal{I}[\![t_n]\!](\mathcal{V}, [b, e])) \\
\mathcal{I}, \mathcal{V}, [b, e] \models \neg A & \text{iff} & \text{not } \mathcal{I}, \mathcal{V}, [b, e] \models A \\
\mathcal{I}, \mathcal{V}, [b, e] \models A \vee B & \text{iff} & \mathcal{I}, \mathcal{V}, [b, e] \models A \text{ or } \mathcal{I}, \mathcal{V}, [b, e] \models B \\
\mathcal{I}, \mathcal{V}, [b, e] \models A \frown B & \text{iff} & \begin{array}{l} \mathcal{I}, \mathcal{V}, [b, m] \models A \text{ and} \\ \mathcal{I}, \mathcal{V}, [m, e] \models B \text{ for some } m \in [b, e] \end{array} \\
\mathcal{I}, \mathcal{V}, [b, e] \models (\exists x \cdot A) & \text{iff} & \mathcal{I}, \mathcal{V}', [b, e] \models A \text{ for some } \mathcal{V}' \text{ } x\text{-equivalent with } \mathcal{V}
\end{array}$$

where $\mathcal{V}'$ is x-equivalent with $\mathcal{V}$ iff $\mathcal{V}'(y) = \mathcal{V}(y)$ for all $y \in GVar \setminus \{x\}$.

Terms The meaning of terms is defined inductively over the syntax by the function $[\![\cdot]\!]$.

$$[\![t]\!] \in \mathbb{I}\mathrm{ntp} \to (\mathbb{V}\mathrm{al} \times \mathbb{I}\mathrm{ntv} \to \mathbb{R})$$

We will use $\mathcal{I}[\![t]\!](\mathcal{V},[b,e])$ instead of the traditional syntax of $[\![t]\!](\mathcal{I})(\mathcal{V},[b,e])$ for application .

$$\begin{array}{lcl}
\mathcal{I}[\![x]\!](\mathcal{V},[b,e]) & = & \mathcal{V}(x) \\
\mathcal{I}[\![f^n(t_1,\ldots,t_n)]\!](\mathcal{V},[b,e]) & = & \overline{f}^n(\mathcal{I}[\![t_1]\!](\mathcal{V},[b,e]),\ldots,\mathcal{I}[\![t_n]\!](\mathcal{V},[b,e])) \\
\mathcal{I}[\![\ell]\!](\mathcal{V},[b,e]) & = & e-b \\
\mathcal{I}[\![v]\!](\mathcal{V},[b,e]) & = & \mathcal{I}(v)([b,e])
\end{array}$$

Truth and validity For given $\mathcal{I}$, $\mathcal{V}$, $[b,e]$ and formula A we say that A is *true*, written $\mathcal{I},\mathcal{V},[b,e] \models A$, if the meaning of the formula, $\mathcal{I}[\![A]\!](\mathcal{V},[b,e])$, is true.

A formula A is *valid*, written $\models A$, iff A is true for all $\mathcal{I}$, $\mathcal{V}$ and $[b,e]$. The formula is *satisfiable* if it is true for some $\mathcal{I}$, $\mathcal{V}$ and $[b,e]$.

2.1.3 Hilbert Style Proof System

In this section we present the proof system as defined in [21]. We call this system $\mathcal{IL}_{\mathcal{H}}$.

$\mathcal{IL}_{\mathcal{H}}$ is based on predicate logic with equality and contains the axioms and rules of this system. The specific axiom and rule sets are immaterial, any may be chosen. It should be noted however that the rule for instantiation of the universal quantifier needs an additional side condition as it will appear in the following.

Syntactic properties In order to state the axioms and rules of $\mathcal{IL}_{\mathcal{H}}$ we define some syntactic properties of formulas and terms: A term is called *flexible* if it contains a temporal variable or ℓ. A formula is flexible if it contains a flexible term or a propositional letter. A term or formula which is not flexible is called *rigid*. A formula is *chop free* if it contains no chop operators ($\frown$) nor modalities defined using chop. A term t is called *free for x in A* if t may be substituted for x in A without free variables in t becoming bound. E.g. y is free for x in $(\forall x \cdot (\forall z \cdot x = z))$ but not in $(\forall x \cdot (\forall y \cdot x = y))$.

We will use the notation $A(x)$ to mean that x occurs free in A and $A(t/x)$ to mean the substitution of t for free occurrences of x in the formula A.

Axioms

$$
\begin{array}{ll}
\mathrm{A0}: & \ell \geq 0 \\[1ex]
\mathrm{A1}: & \begin{array}{l} A \frown B_1 \wedge \neg(A \frown B_2) \Rightarrow A \frown (B_1 \wedge \neg B_2) \\ A_1 \frown B \wedge \neg(A_2 \frown B) \Rightarrow (A_1 \wedge \neg A_2) \frown B \end{array} \\[1ex]
\mathrm{A2}: & (A \frown B) \frown C \Leftrightarrow A \frown (B \frown C) \\[1ex]
\mathrm{R}: & \begin{array}{ll} A \frown B \Rightarrow A & \text{if } A \text{ is a rigid formula} \\ A \frown B \Rightarrow B & \text{if } B \text{ is a rigid formula} \end{array} \\[1ex]
\mathrm{B}: & \begin{array}{ll} (\exists x \cdot A) \frown B \Rightarrow (\exists x \cdot A \frown B) & \text{if } x \text{ is not free in } B \\ A \frown (\exists x \cdot B) \Rightarrow (\exists x \cdot A \frown B) & \text{if } x \text{ is not free in } A \end{array} \\[1ex]
\mathrm{L1}: & \begin{array}{l} (\ell = x) \frown A \Rightarrow \neg((\ell = x) \frown \neg A) \\ A \frown (\ell = x) \Rightarrow \neg(\neg A \frown (\ell = x)) \end{array} \\[1ex]
\mathrm{L2}: & (x \geq 0) \wedge (y \geq 0) \Rightarrow (\ell = x + y \Leftrightarrow (\ell = x) \frown (\ell = y)) \\[1ex]
\mathrm{L3}: & A \Rightarrow A \frown (\ell = 0) \\[1ex]
\mathrm{L4}: & A \Rightarrow (\ell = 0) \frown A \\
\mathrm{Q}: & (\forall x \cdot A(x)) \Rightarrow A(t/x) \quad \begin{array}{l} \text{if } t \text{ is free for } x \text{ in } A \\ \text{and either } t \text{ is rigid or } A(x) \text{ is chop free} \end{array}
\end{array}
$$

Rules

$$
\begin{array}{lll}
\mathrm{MP}: & \text{if } A \text{ and } A \Rightarrow B \text{ then } B & \text{Modus ponens} \\
\mathrm{G}: & \text{if } A \text{ then } (\forall x \cdot A) & \text{Generalization} \\
\mathrm{N1}: & \text{if } A \text{ then } \neg(\neg A \frown B) & \\
\mathrm{N2}: & \text{if } B \text{ then } \neg(A \frown \neg B) & \text{Necessitation} \\
\mathrm{M1}: & \text{if } A \Rightarrow B \text{ then } A \frown C \Rightarrow B \frown C & \\
\mathrm{M2}: & \text{if } A \Rightarrow B \text{ then } C \frown A \Rightarrow C \frown B & \text{Monotonicity}
\end{array}
$$

Deduction and Proof A *deduction* of A from a (possibly infinite) set of assumption formulas Γ is a (finite) sequence of formulas $B_1, \ldots, B_i, \ldots, B_n$ such that $A = B_n$ and

- B_i is an axiom
- or $B_i \in \Gamma$
- or B_i follows by the rules from the formulas $B_1, \ldots, B_{i-1}$

The existence of a deduction of A from Γ is written $\Gamma \vdash A$. A *proof* of A is a deduction of A from the empty set of assumptions. The existence of a proof of A is written $\vdash A$. If $\vdash A$ then A is called a *theorem*.

Proof system properties The proof system is *sound* [21]. This means that all theorems are valid formulas, i.e. that if $\vdash A$ then $\models A$.

The property that all valid formulas are theorems, i.e. if $\models A$ then $\vdash A$, is called *completeness*. $\mathcal{IL}_\mathcal{H}$ is complete wrt. a generalized value and time domain consisting of totally ordered commutative groups [14]. We use a specific totally ordered commutative group namely the real numbers and it is well-known that the real numbers cannot be uniquely identified by any first order theory. For this particular domain $\mathcal{IL}_\mathcal{H}$ is only complete relative to the real numbers.

When doing proofs in $\mathcal{IL}_\mathcal{H}$ it is often convenient to make use of the *deduction theorem* for Duration Calculus [21]:

$$\text{D:} \quad \text{if} \quad \Gamma, A \vdash B \quad \text{then} \quad \Gamma \vdash \Box A \Rightarrow B \qquad \text{Deduction}$$

with the proviso that the rule of generalization (G) is not used on variables free in A.

2.1.4 Sequent Style Proof System

We call this system $\mathcal{IL}_\mathcal{S}$. With a view to supplying tool support for the logic we will be specific about the axiomatic basis and rules of first order predicate logic on which $\mathcal{IL}_\mathcal{S}$ is built. But first we define the concept of a sequent.

Sequent A sequent is a pair of finite sequences of formulas, written $\Gamma \vdash \Delta$. When writing sequences a single formula A stands for the sequence $< A >$ and by Γ, Γ' we mean the concatenation of Γ and Γ'.

The left side of a sequent is called the *antecedent* and formulas on the left are called *antecedent formulas*. The right side of a sequent is called the *consequent* and the formulas *consequent formulas*.

A sequent is valid, written $\models\!\!\models A_1, \ldots, A_n \vdash B_1, \ldots, B_m$, iff $\models A_1 \wedge \cdots \wedge A_n \Rightarrow B_1 \vee \cdots \vee B_m$. If $n = 0$ the antecedent is true and if $m = 0$ then the consequent is false. Note as a special case $\models\!\!\models\vdash B$ iff $\models B$.

First order predicate logic

We use a standard formulation of predicate logic [17]. Notice the added side conditions on the quantifier rules. We write rules with a horizontal line separating the *premises* of the rule above the line from the *conclusion* below the line. Both premises and conclusion are sequent. Axioms are rules with no premises.

Structural rules The structural rules are for thinning, contraction and exchange of subsequences in sequents. Specializations of these rules are structural rules for formula i.e. singular subsequences.

$$\frac{\Gamma, \Gamma'' \vdash \Delta}{\Gamma, \Gamma', \Gamma'' \vdash \Delta}\ \mathrm{T}\vdash \qquad \frac{\Gamma \vdash \Delta, \Delta''}{\Gamma \vdash \Delta, \Delta', \Delta''}\ \vdash \mathrm{T}$$

$$\frac{\Gamma, \Gamma', \Gamma', \Gamma'' \vdash \Delta}{\Gamma, \Gamma', \Gamma'' \vdash \Delta}\ \mathrm{C}\vdash \qquad \frac{\Gamma \vdash \Delta, \Delta', \Delta', \Delta''}{\Gamma \vdash \Delta, \Delta', \Delta''}\ \vdash \mathrm{C}$$

$$\frac{\Gamma, \Gamma', \Gamma'', \Gamma''' \vdash \Delta}{\Gamma, \Gamma'', \Gamma', \Gamma'' \vdash \Delta}\ \mathrm{E}\vdash \qquad \frac{\Gamma \vdash \Delta, \Delta', \Delta'', \Delta'''}{\Gamma \vdash \Delta, \Delta'', \Delta'', \Delta'''}\ \vdash \mathrm{E}$$

Propositional rules

$$\frac{}{A \vdash B}\ \mathrm{Ax}$$ if A and B are syntactically equal up to renaming of bound variables.

$$\frac{\Gamma, A, B, \Gamma' \vdash \Delta}{\Gamma, A \wedge B, \Gamma' \vdash \Delta}\ \wedge\vdash \qquad \frac{\Gamma \vdash \Delta, A, \Delta' \qquad \Gamma \vdash \Delta, B, \Delta'}{\Gamma \vdash \Delta, A \wedge B, \Delta'}\ \vdash\wedge$$

$$\frac{\Gamma, A, \Gamma' \vdash \Delta \qquad \Gamma, B, \Gamma' \vdash \Delta}{\Gamma, A \vee B, \Gamma' \vdash \Delta}\ \vee\vdash \qquad \frac{\Gamma \vdash \Delta, A, B, \Delta'}{\Gamma \vdash \Delta, A \vee B, \Delta'}\ \vdash\vee$$

$$\frac{\Gamma, \Gamma' \vdash \Delta, A \qquad \Gamma, B, \Gamma' \vdash \Delta}{\Gamma, A \Rightarrow B, \Gamma' \vdash \Delta}\ \Rightarrow\vdash \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \Rightarrow B, \Delta}\ \vdash\Rightarrow$$

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

$$\frac{\Gamma \vdash \Delta, A \qquad \Gamma, A \vdash \Delta}{\Gamma \vdash \Delta}\ \mathrm{Cut}$$

Quantifier rules

$$\frac{\Gamma, A(t/x), \Gamma', (\forall x \cdot A) \vdash \Delta}{\Gamma, (\forall x \cdot A), \Gamma' \vdash \Delta}\ \forall\vdash$$ t is any term free for x in A and either t is rigid or A is chop free

$$\frac{\Gamma \vdash \Delta, A, \Delta'}{\Gamma \vdash \Delta, (\forall x \cdot A), \Delta'}\ \vdash\forall$$ x does not occur free in the conclusion

$$\frac{\Gamma \vdash \Delta, A(t/x), \Delta', (\exists x \cdot A)}{\Gamma \vdash \Delta, (\exists x \cdot A), \Delta'}\ \vdash\exists$$ t is any term free for x in A and either t is rigid or A is chop free

$$\frac{\Gamma, A, \Gamma' \vdash \Delta}{\Gamma, (\exists x \cdot A), \Gamma' \vdash \Delta}\ \exists\vdash$$ x does not occur free in the conclusion

Remark We illustrate the need for the provisos on the $\forall \vdash$ and $\vdash \exists$ rules by two examples.

Consider the first proviso and the formula $A(x) \hat{=} (\exists y \cdot y > x)$ and the term $t \hat{=} y$ with $A(t/x) = (\exists y \cdot y > y)$. t is *not* free for x is A. Without the proviso we can construct the following derivation with an invalid root:

$$\dfrac{\dfrac{}{(\exists y \cdot y > y), (\forall x \cdot (\exists y \cdot y > x)) \vdash (\exists y \cdot y > y)}\, Ax}{(\forall x \cdot (\exists y \cdot y > x)) \vdash (\exists y \cdot y > y)}\, \forall \vdash$$

To see the need for the second proviso on the $\forall \vdash$ and $\vdash \exists$ rules consider the formula $A(x)$ and flexible term ℓ:

$$\begin{array}{lcl} A(x) & \hat{=} & (\ell = x) \frown (\ell = x) \Rightarrow \ell = 2x \\ A(\ell/x) & = & (\ell = \ell) \frown (\ell = \ell) \Rightarrow \ell = 2\ell \end{array}$$

Note that $A(x)$ is not chop free and consider the instance:

$$\dfrac{(\ell = \ell) \frown (\ell = \ell) \Rightarrow \ell = 2\ell \vdash \ell = 0}{(\forall x \cdot (\ell = x) \frown (\ell = x) \Rightarrow \ell = 2x) \vdash \ell = 0}\, \forall \vdash$$

The premise is valid because $A(\ell/x)$ is satisfiable only for $\ell = 0$ but the conclusion is invalid. $\Box$

Equality For reference we list the standard axioms of equality and make an important remark.

$$\dfrac{}{\Gamma \vdash a = b, \Delta}\, \texttt{refl} \qquad \begin{array}{l} a \text{ and } b \text{ are syntactically equal} \\ \text{up to renaming of bound variables} \end{array}$$

$$\dfrac{}{a_1 = b_1, \ldots, a_n = b_n \vdash f(a_1, \ldots, a_n) = f(b_1, \ldots, b_n)}\, \texttt{fun\_repl}$$

$$\dfrac{}{a_1 = b_1, \ldots, a_n = b_n, G(a_1, \ldots, a_n) \vdash G(b_1, \ldots, b_n)}\, \texttt{pred\_repl}$$

A consequence of these axioms is that $=$ is an equivalence relation i.e. reflexive, symmetric and transitive ([20] page 108).

Remark The axioms above have been taken from [20] where they appear in an exposition of first order predicate logic. In such a logic the axioms have as a consequence the convenient rule of *substitution of equals for equals.*

$$\frac{}{a = b, A(a) \vdash A(b)}\ \texttt{eq\_for\_eq}$$

Such a rule is *not* sound for $\mathcal{IL}_{\mathcal{S}}$. Consider the rule instance below where 3 is substituted for ℓ throughout. This instance is clearly not valid.

$$\frac{}{\ell = 3, \ell = 3 \Rightarrow (\ell = 2) \frown (\ell = 1) \vdash 3 = 3 \Rightarrow (3 = 2) \frown (3 = 1)}$$

In $\mathcal{IL}_{\mathcal{S}}$ the equality need to be boxed for the rule to be sound:

$$\frac{}{\Box(a = b), A(a) \vdash A(b)}\ \texttt{subst}$$

Now the instance is valid:

$$\frac{}{\Box(\ell = 3), \ell = 3 \Rightarrow (\ell = 2) \frown (\ell = 1) \vdash 3 = 3 \Rightarrow (3 = 2) \frown (3 = 1)}$$

because $\Box(\ell = 3)$ is unsatisfiable. $\Box$

Real Arithmetic

For reference we give here the field and order axioms for real numbers. For a precise characterization of the real numbers, also the axiom of completeness must be included. We make no use of this axiom and leave it out.

Field axioms

$$\frac{}{\vdash x + y = y + x}\ \texttt{commutative\_add}$$

$$\frac{}{\vdash x + (y + z) = (x + y) + z}\ \texttt{associative\_add}$$

$$\frac{}{\vdash x + 0 = x}\ \texttt{identity\_add}$$

$$\frac{}{\vdash x + -x = 0}\ \texttt{inverse\_add}$$

$$\frac{}{\vdash x - y = x + -y}\ \texttt{minus\_add}$$

$$\frac{}{\vdash x * y = y * x}\ \texttt{commutative\_mult}$$

$$\frac{}{\vdash x * (y * z) = (x * y) * z}\ \texttt{associative\_mult}$$

$$\frac{}{\vdash 1 * x = x}\ \texttt{identity\_mult}$$

$$\frac{}{\vdash \neg(x = 0) \Rightarrow x * (1/x) = 1}\ \texttt{inverse\_mult}$$

$$\frac{}{\vdash \neg(x = 0) \Rightarrow y/x = y * (1/x)}\ \texttt{div\_def}$$

$$\frac{}{\vdash x * (y + z) = (x * y) + (x * z)}\ \texttt{distributive}$$

Order axioms

$$\frac{}{\vdash x > 0 \wedge y > 0 \Rightarrow x + y > 0}\ \texttt{posreal\_add\_closed}$$

$$\frac{}{\vdash x > 0 \wedge y > 0 \Rightarrow x * y > 0}\ \texttt{posreal\_mult\_closed}$$

$$\frac{}{\vdash x > 0 \Rightarrow \neg(-x > 0)}\ \texttt{posreal\_neg}$$

$$\frac{}{\vdash x > 0 \vee x = 0 \vee 0 > x}\ \texttt{trichotomy}$$

Interval Logic

The axioms of $\mathcal{IL}_{\mathcal{H}}$ are simply given a sequent formulation in $\mathcal{IL}_{\mathcal{S}}$ but are otherwise unchanged. The rules of $\mathcal{IL}_{\mathcal{S}}$ however are different. Due to the presence of antecedent and consequent formulas the premise of the monotonicity rules have to be strengthened. Also the rules of necessitation is combined into one rule.

Axioms of $\mathcal{IL}_\mathcal{S}$

$$\frac{}{\vdash \ell \geq 0}\ \texttt{A0}$$

$$\frac{}{\vdash A \frown B_1 \wedge \neg(A \frown B_2) \Rightarrow A \frown (B_1 \wedge \neg B_2)}\ \texttt{A1a}$$

$$\frac{}{\vdash A_1 \frown B \wedge \neg(A_2 \frown B) \Rightarrow (A_1 \wedge \neg A_2) \frown B}\ \texttt{A1b}$$

$$\frac{}{\vdash A \frown (B \frown C) \Leftrightarrow (A \frown B) \frown C}\ \texttt{A2}$$

$$\frac{}{\vdash x \geq 0 \wedge y \geq 0 \Rightarrow (\ell = x + y \Leftrightarrow (\ell = x) \frown (\ell = y))}\ \texttt{L2}$$

$$\frac{}{\vdash A \frown B \Rightarrow A}\ \texttt{Ra} \quad A \text{ is rigid}$$

$$\frac{}{\vdash A \frown B \Rightarrow B}\ \texttt{Rb} \quad B \text{ is rigid}$$

$$\frac{}{\vdash (\exists x \cdot A) \frown B \Rightarrow (\exists x \cdot A \frown B)}\ \texttt{Ba} \quad x \text{ does not occur free in } B$$

$$\frac{}{\vdash A \frown (\exists x \cdot B) \Rightarrow (\exists x \cdot A \frown B)}\ \texttt{Bb} \quad x \text{ does not occur free in } B$$

$$\frac{}{\vdash (\ell = x) \frown A \Rightarrow \neg((\ell = x) \frown \neg A)}\ \texttt{L1a} \qquad \frac{}{\vdash A \frown (\ell = x) \Rightarrow \neg(\neg A \frown (\ell = x))}\ \texttt{L1b}$$

$$\frac{}{\vdash (\ell = 0) \frown A \Leftrightarrow A}\ \texttt{L3} \qquad \frac{}{\vdash A \frown (\ell = 0) \Leftrightarrow A}\ \texttt{L4}$$

Rules of $\mathcal{IL}_\mathcal{S}$ Let $\Gamma^\Box$ be the formulas in Γ of the form $\Box A$ and let $\Delta^\Diamond$ be the formulas in Δ of the form $\Diamond A$.

$$\frac{\Gamma \vdash \Box(A \Rightarrow B), \Delta}{\Gamma \vdash A \frown C \Rightarrow B \frown C, \Delta}\ \texttt{monotone\_chop} \qquad \frac{\Gamma \vdash \Box(A \Rightarrow B), \Delta}{\Gamma \vdash C \frown A \Rightarrow C \frown B, \Delta}\ \texttt{chop\_monotone}$$

$$\frac{\Gamma^\Box \vdash A, \Delta^\Diamond}{\Gamma \vdash \Box A, \Delta}\ \texttt{boxR}$$

Remark Removing the box in the premise of the monotonicity rules would produce the rule:

$$\frac{\Gamma \vdash A \Rightarrow B, \Delta}{\Gamma \vdash A \frown C \Rightarrow B \frown C, \Delta}\ \texttt{mc}$$

Such a rule is *unsound*. To see this consider the instance:

$$\frac{l = 2 \vdash l = 1 \Rightarrow l = 47}{l = 2 \vdash l = 1 \frown l = 1 \Rightarrow l = 47 \frown l = 1}$$

While the premise is valid the conclusion is clearly not. □

Deduction and Proof

A *deduction* in the sequent style logic is a finite *tree* the nodes of which are sequents such that a node is either a leaf node or a node and its children is an instantiation of a rule. A *proof* is a deduction where all leafs are instances of axioms.

The root of a deduction is called the *conclusion*. The set of leaves Σ of a deduction is called the *assumptions*.

A proof with conclusion σ is called a *proof of* σ and σ is called *provable*. If $\vdash A$ is a provable sequent then A is called a *theorem*.

Derived Rules

The above rules are basic for the sequent style formulation of DC. From these rules it is possible to derive further rules which may be more appropriate for actual use.

$$\frac{\Gamma^{\Box}, A \vdash \Delta^{\Diamond}}{\Gamma, \Diamond A \vdash \Delta}\ \texttt{diaL}$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, \Box A \vdash \Delta}\ \texttt{boxL} \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash \Diamond A, \Delta}\ \texttt{diaR}$$

$$\frac{\Gamma \vdash \Box(A \Rightarrow B), \Delta \qquad \Gamma, A \frown C, B \frown C \vdash \Delta}{\Gamma, A \frown C \vdash \Delta}\ \texttt{mono\_chop}$$

$$\frac{\Gamma \vdash \Box(A \Rightarrow B), \Delta \qquad \Gamma, C \frown A, C \frown B \vdash \Delta}{\Gamma, C \frown A \vdash \Delta}\ \texttt{chop\_mono}$$

Proof System Properties

In this section we will investigate the properties of the sequent style proof system just presented and we will show that it is just as powerful as the Hilbert style system in section 2.1.3.

Recall from section 2.1.3 that $\mathcal{IL}_{\mathcal{H}}$ is sound and complete on a generalized domain. We will investigate whether similar properties hold for $\mathcal{IL}_{\mathcal{S}}$.

Soundness By definition $\mathcal{IL}_{\mathcal{S}}$ is sound if the theorems of $\mathcal{IL}_{\mathcal{S}}$ are all valid. To prove this is an induction on the size of the proof tree of the theorem. The induction, as always, has a base and an inductive case.

The base case consists of showing that all axioms or proof trees of size 1 are valid. We do this implicitly by observing that $\mathcal{IL}_{\mathcal{S}}$ and $\mathcal{IL}_{\mathcal{H}}$ share the same axioms and that the axioms of $\mathcal{IL}_{\mathcal{H}}$ are valid since $\mathcal{IL}_{\mathcal{H}}$ is sound.

Assuming that theorems with proof trees of size n are all valid the inductive case consists of proving that theorems with proof trees of size $n+1$ will also be valid. Essentially this boils down to showing that the proof rules of $\mathcal{IL}_{\mathcal{S}}$ all preserve validity, meaning that if the premises of a rule are valid then the conclusion will also be valid.

We accept without proof the soundness of the structural, propositional and quantifier rules. This leaves the soundness of the rules `monotone_chop`, `chop_monotone` and `boxR` to be investigated. We have encoded validity of sequents and formulas of IL in PVS and performed the soundness proofs using this encoding (see appendix E) . We conclude $\mathcal{IL}_{\mathcal{S}}$ is sound.

Deductive power We want $\mathcal{IL}_{\mathcal{S}}$ to be as powerful a proof system as $\mathcal{IL}_{\mathcal{H}}$, meaning that all theorems of $\mathcal{IL}_{\mathcal{H}}$ should also be theorems of $\mathcal{IL}_{\mathcal{S}}$. Fortunately this turns out to be the case and we demonstrate it by an induction proof over the length of proofs in $\mathcal{IL}_{\mathcal{H}}$.

The base case are proofs in $\mathcal{IL}_{\mathcal{H}}$ of length 1, ie. axioms. Since $\mathcal{IL}_{\mathcal{H}}$ and $\mathcal{IL}_{\mathcal{S}}$ share axioms it is immediate that a proof also exists in $\mathcal{IL}_{\mathcal{S}}$.

For the inductive case we assume that theorems of $\mathcal{IL}_{\mathcal{H}}$ with proofs of length n have proofs in $\mathcal{IL}_{\mathcal{S}}$. We want to prove that theorems of $\mathcal{IL}_{\mathcal{H}}$ with proofs of length $n+1$ also have proofs in $\mathcal{IL}_{\mathcal{S}}$. We do this by demonstrating how to build such proofs.

Recall that a proof in $\mathcal{IL}_{\mathcal{H}}$ is a sequence of formulas. Consider a proof of length n and note that all formulas in such a proof are theorems. Indeed they are theorems with proofs no longer than n. By the assumption above these theorems all have proofs in $\mathcal{IL}_{\mathcal{S}}$.

Proofs in $\mathcal{IL}_{\mathcal{H}}$ of length $n+1$ are constructed by extending some proof of length n with one more formula B, either an axiom or an inference using some rule.

In case the extension is an axiom clearly this axiom has a proof in $\mathcal{IL}_{\mathcal{S}}$. If the extension is by inference using some rule R we will construct a proof in $\mathcal{IL}_{\mathcal{S}}$ of the formula B from the proofs (in $\mathcal{IL}_{\mathcal{S}}$) of theorems occurring earlier in the proof sequence.

$\mathcal{IL}_{\mathcal{H}}$ has 8 inference rules. We can make such a construction for each rule. For the construction we first derive rules in $\mathcal{IL}_{\mathcal{S}}$ to mimic the rules in $\mathcal{IL}_{\mathcal{H}}$. These rules are:

$$\frac{\vdash A \qquad \vdash A \Rightarrow B}{\vdash B}\ \texttt{MP}$$

$$\frac{\vdash A(x)}{\vdash (\forall x \cdot A(x))}\ \texttt{G}$$

$$\frac{\vdash A}{\vdash \neg((\neg A) \frown B)}\ \texttt{N1} \qquad \frac{\vdash A}{\vdash \neg(B \frown (\neg A))}\ \texttt{N2}$$

$$\frac{\vdash A \Rightarrow B}{\vdash A \frown C \Rightarrow B \frown C}\ \texttt{M1} \qquad \frac{\vdash A \Rightarrow B}{\vdash C \frown A \Rightarrow C \frown B}\ \texttt{M2}$$

It is important to note that we can derive these rules in the system $\mathcal{IL}_{\mathcal{S}}$ itself. In fact we have used the tool presented in chapter 6 to do this (see appendix C.4).

Now for the construction. There are 6 similar cases, we show only one.

$R = MP$ In this case extension is by the *MP* rule and B is derived from some A and $A \Rightarrow B$ occurring in the proof of B. By assumption both A and $A \Rightarrow B$ have proofs in $\mathcal{IL}_{\mathcal{S}}$. We now simply combine the proofs using the derived `MP` rule to construct a proof in $\mathcal{IL}_{\mathcal{S}}$ of the sequent $\vdash A$. This proof makes the formula A a theorem of $\mathcal{IL}_{\mathcal{S}}$.

Completeness Recall that $\mathcal{IL}_{\mathcal{H}}$ is complete wrt. a value and time domain consisting of totally ordered commutative groups. The result also holds for $\mathcal{IL}_{\mathcal{S}}$. This is a consequence of $\mathcal{IL}_{\mathcal{S}}$ being as deductively powerful as $\mathcal{IL}_{\mathcal{H}}$.

2.2 Duration Calculus

The Duration Calculus builds on Interval Logic by imposing a structure on the temporal variables. These are now durations of boolean valued functions of time over an interval.

2.2.1 Syntax

The syntax of DC reflects the added structure by defining a new syntactic class, the *state formula*.

The state formulas of DC are generated inductively from the set of *state variable* symbols:

$$s \in SVar \quad \text{State variables}$$

The inductive definition is given by this syntax:

$$P, Q \ ::= \ \mathit{ff} \mid tt \mid s \mid \neg P \mid P \vee Q \quad \text{State formulas}$$

The remaining logical connectives of state formulas are defined using the usual abbreviations:

$$\begin{array}{lcl} P \wedge Q & \hat{=} & \neg(\neg P \vee \neg Q) \\ P \Rightarrow Q & \hat{=} & \neg P \vee Q \\ P \Leftrightarrow Q & \hat{=} & (P \Rightarrow Q) \wedge (P \Rightarrow Q) \end{array}$$

The syntax of terms changes slightly. Temporal variables are replaced by *duration terms* $\int P$. The length ℓ is expressible as an abbreviation and thus disappears from the basic syntax.

$$t \ ::= \ x \mid f^n(t_1, \ldots, t_n) \mid \int P \quad \text{Terms}$$

$$\begin{array}{lcl} \ell & \hat{=} & \int tt \\ \lceil P \rceil & \hat{=} & \int P = \ell \wedge \ell > 0 \\ \lceil\, \rceil & \hat{=} & \ell = 0 \end{array}$$

The formula $\lceil P \rceil$ is called a *state predicate* and the formula $\lceil\, \rceil$ is called a *point*.

2.2.2 Semantics

State variables and propositional letters The meaning of state variables and propositional letters is captured collectively by an *interpretation* $\mathcal{I}$.

$$\mathcal{I} \in \mathbb{I}\mathrm{ntp} = \begin{pmatrix} SVar \\ \cup \\ PLetter \end{pmatrix} \to \begin{pmatrix} \mathbb{T}\mathrm{ime} \stackrel{\mathrm{fv}}{\to} \mathbb{B} \\ \cup \\ \mathbb{I}\mathrm{ntv} \to \mathbb{B} \end{pmatrix}$$

with the restriction that $\mathcal{I}(s) \in \mathbb{T}\mathrm{ime} \stackrel{\mathrm{fv}}{\to} \mathbb{B}$ and $\mathcal{I}(X) \in \mathbb{I}\mathrm{ntv} \to \mathbb{B}$. $\mathbb{T}\mathrm{ime} \stackrel{\mathrm{fv}}{\to} \mathbb{B}$ is the set of *finitely variable* state functions i.e. boolean valued functions of time that have only finitely many discontinuity points in any given interval.

Terms The meaning of duration terms is defined as:

$$\begin{array}{lcl} \mathcal{I}[\![\int P]\!](\mathcal{V}, [b, e]) & = & \int_b^e \mathcal{X}(I[\![P]\!](t))\,dt \\ \mathcal{X}(b) & = & \text{if } b \text{ then } 1 \text{ else } 0 \end{array}$$

where $\int_b^e f(t)dt$ is the definite integral of f from b to e and $\mathcal{X}(\lfloor)$ is the indicator function for b. With the restriction to finitely variable state functions the Riemann integral is always well-defined.

State formulas The meaning of state formulas is defined inductively over the syntax by the function $[\![\cdot]\!]$.

$$[\![P]\!] \in \mathbb{I}\mathrm{ntp} \rightarrow (\mathbb{T}\mathrm{ime} \rightarrow \mathbb{B})$$

We will use $\mathcal{I}[\![P]\!]t$ instead of the more traditional syntax for application $[\![P]\!](\mathcal{I})(t)$, where t is a time point.

$$\begin{array}{lcl}
\mathcal{I}[\![\mathit{ff}]\!]t & = & \mathit{false} \\
\mathcal{I}[\![\mathit{tt}]\!]t & = & \mathit{true} \\
\mathcal{I}[\![s]\!]t & = & \mathcal{I}(s)(t) \\
\mathcal{I}[\![\neg P]\!]t & = & \text{not } \mathcal{I}[\![P]\!](t) \\
\mathcal{I}[\![P \vee Q]\!]t & = & \mathcal{I}[\![P]\!](t) \text{ or } \mathcal{I}[\![Q]\!](t)
\end{array}$$

2.2.3 Hilbert Style Proof System

The Hilbert style proof system for DC extends $\mathcal{IL}_{\mathcal{H}}$ with the following axioms and rules. We call the new system $\mathcal{DC}_{\mathcal{H}}$.

Axioms

$$\begin{array}{ll}
\mathrm{DCA1}: & \int \mathit{ff} = 0 \\
\mathrm{DCA2}: & \int \mathit{tt} = \ell \\
\mathrm{DCA3}: & \int P \geq 0 \\
\mathrm{DCA4}: & \int P + \int Q = \int (P \vee Q) + \int (P \wedge Q) \\
\mathrm{DCA5}: & (\int P = x) \frown (\int P = y) \Rightarrow \int P = x + y \\
\mathrm{DCA6}: & \int P = \int Q \quad \text{if } P \Leftrightarrow Q \text{ holds in propositional logic}
\end{array}$$

Rules

$$\begin{array}{llllll}
\mathit{IR}1: & \text{if} \quad H(\lceil\,\rceil) & \text{and} & H(X) \Rightarrow H(X \vee X \frown \lceil P \rceil \vee X \frown \lceil \neg P \rceil) & \text{then} & H(\mathit{true}) \\
\mathit{IR}2: & \text{if} \quad H(\lceil\,\rceil) & \text{and} & H(X) \Rightarrow H(X \vee \lceil P \rceil \frown X \vee \lceil \neg P \rceil \frown X) & \text{then} & H(\mathit{true})
\end{array}$$

When doing proofs in $\mathcal{DC}_{\mathcal{H}}$ it is often convenient to make use of the *deduction theorem* for Duration Calculus [21]:

$$\text{D:} \qquad \text{if} \quad \Gamma, A \vdash B \quad \text{then} \quad \Gamma \vdash \Box A \Rightarrow B \qquad \text{Deduction}$$

provided that the rule of generalization (G) is not used on variables free in A and that predicate letters used in the application of induction rules do not occur in A.

Proof System Properties

It can be proved that $\mathcal{DC}_\mathcal{H}$ is sound [21].

We cannot hope for $\mathcal{DC}_\mathcal{H}$ to be complete for the same reason that $\mathcal{IL}_\mathcal{H}$ is not complete in the domain of the real numbers. However, consider the set IL of all valid formulas of $\mathcal{IL}_\mathcal{H}$ and the set IL_{dc} of all instantiations of formulas from IL replacing temporal variables with duration terms. All formulas of IL_{dc} will be valid. Now $\mathcal{DC}_\mathcal{H}$ is *relative complete* [21] in the sense that for any formula A if $\models A$ then $IL_{dc} \vdash A$.

2.2.4 Sequent Style Proof System

We call this system $\mathcal{DC}_\mathcal{S}$. It has the following axioms and rules:

$$\frac{}{\vdash \int ff = 0}\ \texttt{DCAX1} \qquad \frac{}{\vdash \int tt = l}\ \texttt{DCAX2}$$

$$\frac{}{\vdash \int s \geq 0}\ \texttt{DCAX3} \qquad \frac{}{\vdash \int s_1 + \int s_2 = \int (s_1 \vee s_2) + \int (s_1 \wedge s_2)}\ \texttt{DCAX4}$$

$$\frac{}{\vdash (\int s = x) \frown (\int s = y) \Rightarrow \int s = x + y}\ \texttt{DCAX5}$$

$$\frac{}{\vdash \int s_1 = \int s_2}\ \texttt{DCAX6} \quad \text{iff } s_1 \Leftrightarrow s_2 \text{ holds in propositional logic}$$

$$\frac{\Gamma \vdash R(\lceil\,\rceil), \Delta \qquad \Gamma \vdash (R(X) \Rightarrow R(X \vee \lceil s \rceil \frown X \vee \lceil \neg s \rceil \frown X), \Delta}{\Gamma \vdash R(true), \Delta}\ \texttt{chop\_induct}$$

$$\frac{\Gamma \vdash R(\lceil\,\rceil), \Delta \qquad \Gamma \vdash (R(X) \Rightarrow R(X \vee X \frown \lceil s \rceil \vee X \frown \lceil \neg s \rceil), \Delta}{\Gamma \vdash R(true), \Delta}\ \texttt{induct\_chop}$$

Remark In [45] the induction rules are formulated with a heavy use of boxing:

$$\frac{\Gamma \vdash \Box R(\lceil\,\rceil), \Delta \qquad \Gamma \vdash \Box(\Box R(X) \Rightarrow R(X \vee \lceil s \rceil \frown X \vee \lceil \neg s \rceil \frown X), \Delta}{\Gamma \vdash \Box R(true), \Delta}$$

It turns out that this rule is equivalent with `induct_chop`, the rules can be derived from each other. This is not hard to see when remembering two well-known theorems of Interval Logic:

$$\Box A \Leftrightarrow \Box\Box A$$
$$\Box(A \Rightarrow B) \Rightarrow (\Box A \Rightarrow \Box B)$$

$\Box$

Proof System Properties

In this section we will investigate the properties of the sequent style proof system just presented and we will relate it to the Hilbert style system $\mathcal{DC}_\mathcal{H}$.

Recall that $\mathcal{DC}_\mathcal{H}$ is both sound and relative complete. We will investigate whether similar properties hold for $\mathcal{DC}_\mathcal{S}$.

Soundness The approach to proving soundness is the same used for $\mathcal{IL}_\mathcal{S}$: Prove that the new axioms are valid and that the new rules preserve validity.

We have done these proofs in the theorem prover PVS using an encoding of validity of sequents and formulas of DC (see Appendix E.2). We conclude that $\mathcal{DC}_\mathcal{S}$ is sound.

Deductive power We want $\mathcal{DC}_\mathcal{S}$ to be as powerful a proof system as $\mathcal{DC}_\mathcal{H}$. Fortunately this turns out to be the case and we demonstrate it the same way as for $\mathcal{IL}_\mathcal{S}$.

For the inductive case we derive the rules (see appendix C):

$$\frac{\vdash H(\lceil\,\rceil) \quad \vdash H(X) \Rightarrow H(X \vee (X \frown \lceil P \rceil) \vee (X \frown \lceil \neg P \rceil))}{\vdash H(\mathit{true})}\ \mathtt{IR1}$$

$$\frac{\vdash H(\lceil\,\rceil) \quad \vdash H(X) \Rightarrow H(X \vee (\lceil P \rceil \frown X) \vee (\lceil \neg P \rceil \frown X))}{\vdash H(\mathit{true})}\ \mathtt{IR2}$$

Remark Is $\mathcal{DC}_\mathcal{S}$ *more* powerful than $\mathcal{DC}_\mathcal{H}$, does $\mathcal{DC}_\mathcal{S}$ have theorems which are not theorems of $\mathcal{DC}_\mathcal{H}$? We have not investigated this question in depth but conjecture that this is *not* the case. We base this on the following considerations.

First of all we have demonstrated that $\mathcal{DC}_\mathcal{S}$ is sound. This limits the set of possible theorems of $\mathcal{DC}_\mathcal{S}$ to valid formulas.

Secondly $\mathcal{DC}_\mathcal{H}$ is relative complete. Although this leaves room for $\mathcal{DC}_\mathcal{S}$ to be stronger (and still sound) we see no reason to think $\mathcal{DC}_\mathcal{S}$ would not be equally limited. Both systems suffer from an inadequate characterization of the real numbers which is at the root of the relative completeness of $\mathcal{DC}_\mathcal{H}$.
$\square$

Completeness We cannot hope that $\mathcal{DC}_\mathcal{S}$ is complete for the same reasons that $\mathcal{DC}_\mathcal{H}$ is not complete. However, because we have demonstrated that $\mathcal{DC}_\mathcal{S}$ is at least as deductively powerful as $\mathcal{DC}_\mathcal{H}$ we can infer that $\mathcal{DC}_\mathcal{S}$ is relative complete because $\mathcal{DC}_\mathcal{H}$ is.

2.3 Typed States

Until now we have stayed true to the exposition by Hansen and Zhou [21] and only considered boolean valued states. However in system modeling it is convenient to allow richer state domains like natural numbers, lists and real numbers.

In this section we will relax the confinement to boolean states and will defined the necessary syntactical and semantical extensions. In doing so we follow Ravn [40].

2.3.1 Types

A type T is a non-empty set of values equipped with a topology. Requiring a topology allow us to speak about continuity of states in section 2.4.

We consider as types the real numbers $\mathbb{R}$, discrete countable types like the boolean values, natural numbers and finite sequences of simpler types and product types.

The topology of the reals is the usual topology induced by the Euclidean metric and the topology of discrete types is the topology where all subsets are open. For product types the topology is the product topology.

2.3.2 Typed Semantic Domains

We will assign to each state variable s its type $\mathit{Type}(s)$ and model the meaning of s as functions from time to the set $\mathit{Type}(s)$. This means the the domain of interpretations $\mathcal{I}$ change slightly such that $\mathcal{I}(s) \in \mathbb{T}\mathrm{ime} \xrightarrow{\mathrm{fv}} \mathit{Type}(x)$.

Global variables x are assigned the type $\mathit{Type}(x)$ and takes on values in this set. The domain of valuations $\mathbb{V}\mathrm{al}$ is restricted such that $\mathcal{V}(x) \in \mathit{Type}(x)$. Function and relational symbols are assigned a signature

$$
\begin{array}{lcl}
f^n & : & T_1 \times \cdots \times T_n \rightarrow T \\
G^n & : & T_1 \times \cdots \times T_n \rightarrow \mathbb{B}
\end{array}
$$

and take values $\overline{f^n}$ and $\overline{G^n}$ in these domains.

2.3.3 Syntax

To exploit the richer type system we generalize the concept of state formulas

$$
P, Q \quad ::= \quad \mathit{ff} \mid \mathit{tt} \mid s \mid \neg P \mid P \vee Q \quad \text{State formulas}
$$

and introduce *state terms* as a syntactic category to replace it:

$$
st \quad ::= \quad x \mid s \mid f^n(st_1, \ldots, st_n) \quad \text{State terms}
$$

Notice that global variables are now allowed as state terms.

The syntax still captures state formulas if we introduce function symbols $f_{tt}, f_{ff}, f_{\neg}, f_{\vee}$ to represent the constants tt, ff and the operators $\neg$ and $\vee$, respectively. State formulas are now state terms of type boolean and we continue to use P and Q to denote state formulas.

2.3.4 Type System

With the new syntax and typed variables we can compose terms and state terms from function symbols which signatures do not match. To avoid considering such terms we define the sets of well-formed formulas and henceforth only consider well-formed syntax.

We start be defining well-formed state terms. A state term is well-formed iff it is assigned a type by the function $\mathcal{T}(st)$ defined inductively over the syntax:

$$
\begin{array}{lcl}
\mathcal{T}(x) & = & \mathit{Type}(x) \\
\mathcal{T}(s) & = & \mathit{Type}(s) \\
\mathcal{T}(f^n(st_1, \ldots, st_n)) & = & T \text{ iff } \begin{array}{l} f^n : T_1 \times \cdots \times T_n \to T \text{ and} \\ \mathcal{T}(st_i) = T_i \text{ for } i \in \{1 \ldots n\} \end{array}
\end{array}
$$

Likewise for terms. A term is well-formed iff it is assigned a type by the function $\mathcal{T}(t)$ defined inductively over the syntax:

$$
\begin{array}{lcl}
\mathcal{T}(x) & = & \mathit{Type}(x) \\
\mathcal{T}(f^n(t_1, \ldots, t_n)) & = & T \text{ iff } f^n : T_1 \times \cdots \times T_n \to T \text{ and } \mathcal{T}(t_i) = T_i \text{ for } i \in \{1 \ldots n\} \\
\mathcal{T}(\int P) & = & \mathbb{R} \text{ iff } \mathcal{T}(P) = \mathbb{B}
\end{array}
$$

And finally for formulas. A formula A is well-formed iff for any atomic formula $G^n(t_1, \ldots, t_n)$ in A with $G^n : T_1 \times \cdots \times T_n \to \mathbb{B}$, $\mathcal{T}(t_i) = T_i$ for $i \in \{1 \ldots n\}$

2.3.5 Semantics

We define the semantics of state term st inductively over the syntax by the function $[\![st]\!] \in \mathbb{I}\mathrm{ntp} \to \mathbb{V}\mathrm{al} \to \mathcal{T}(st)$. We write $\mathcal{I}[\![st]\!](\mathcal{V}, t)$ to mean $[\![st]\!](\mathcal{I})(\mathcal{V})(t)$ for a time point t.

$$
\begin{array}{lcl}
\mathcal{I}[\![x]\!](\mathcal{V}, t) & = & \mathcal{V}(x) \\
\mathcal{I}[\![s]\!](\mathcal{V}, t) & = & \mathcal{I}(s)(t) \\
\mathcal{I}[\![f^n(st_1, \ldots, st_n)]\!](\mathcal{V}, t) & = & \overline{f^n}(\mathcal{I}[\![st_1]\!](\mathcal{V}, t), \ldots, \mathcal{I}[\![st_n]\!](\mathcal{V}, t))
\end{array}
$$

To keep the correct interpretation of state formulas we assume the following meaning of the function symbols $f_{tt}, f_{ff}, f_{\neg}$ and $f_{\vee}$:

$$
\begin{array}{lcl}
\overline{f_{tt}} & = & \mathit{true} \\
\overline{f_{ff}} & = & \mathit{false} \\
\overline{f_{\neg}}(b) & = & \text{not } b \\
\overline{f_{\vee}}(b_1, b_2) & = & b_1 \text{ or } b_2
\end{array}
$$

2.3.6 Proof System

These new semantics of state terms maintain the soundness of axioms and rules of DC [40] with the exception that we need to strengthen the side condition of rules of quantifier instantiation to disallow replacing the quantified variable with a non-rigid term under a duration operator.

$$\frac{\Gamma, A(t/x), \Gamma', (\forall x \cdot A) \vdash \Delta}{\Gamma, (\forall x \cdot A), \Gamma' \vdash \Delta} \ \forall\vdash$$

t is any term free for x in A and either t is rigid or A is chop free and x does not occur in any duration term of A

$$\frac{\Gamma \vdash \Delta, A(t/x), \Delta', (\exists x \cdot A)}{\Gamma \vdash \Delta, (\exists x \cdot A), \Delta'} \ \vdash\exists$$

t is any term free for x in A and either t is rigid or A is chop free and x does not occur in any duration term of A

We generalize the induction rules to induct on states s of any type, not just boolean. We are requiring that the type is equipped with equality $=$.

$$\frac{\Gamma \vdash R(\lceil\,\rceil), \Delta \qquad \Gamma \vdash (R(X) \Rightarrow R(X \vee (\exists x \cdot \lceil s = x \rceil \frown X)), \Delta}{\Gamma \vdash R(true), \Delta} \ \texttt{typed\_chop\_induct}$$

$$\frac{\Gamma \vdash R(\lceil\,\rceil), \Delta \qquad \Gamma \vdash (R(X) \Rightarrow R(X \vee (\exists x \cdot X \frown \lceil s = x \rceil)), \Delta}{\Gamma \vdash R(true), \Delta} \ \texttt{typed\_induct\_chop}$$

The `DCAX6` rule links equality of durations with logical equivalence of propositional formulas. With the generalization to typed states we allow (quantifier free) first order terms under the duration operator. Therefore it is natural that we also generalize the `DCAX6` rule suitably to link duration equality to logical equivalence of first order formulas.

A suggestion for such an generalization would be to simply generalize the side-condition:

$$\frac{}{\vdash \int s_1 = \int s_2} \qquad \text{iff } s_1 \Leftrightarrow s_2 \text{ holds in predicate logic}$$

But we can generalize this rule even further maintaining assumptions about the global variables in s_1 and s_2. Let $\vdash_{PL}$ denote the deduction relation of Predicate Logic and let Γ^{RC} denote the sequence of *rigid chop free* formulas of Γ. We now consider generalizing the rule to:

$$\frac{\Gamma^{RC} \vdash_{PL} s_1 \Leftrightarrow s_2, \Delta^{RC}}{\Gamma \vdash \int s_1 = \int s_2, \Delta}$$

The rule still links sequents of two different logics: Predicate Logic and DC. With an eye towards encoding the rule in a theorem prover as we will do in chapter 6 this is a drawback. We would prefer a rule expressed entirely in DC and will present such a rule in chapter 6.

2.4 Extended Duration Calculus

The Extended Duration Calculus (EDC) was introduced by Zhou, Ravn and Hansen in [11] and investigated in detail by Ravn in [40] from where we derive this treatment.

DC is restricted in its model to only considering finitely variable functions of time. EDC relaxes this restriction to a larger set of models containing for example continuous real valued functions of time. This allows EDC to capture systems normally dealt with in the field of control theory.

In DC observing a state variable is done by measuring the duration ($\int P$) of a boolean state formula over an interval. EDC contains the duration operator but also defines the integral ($\int se$) of a real valued state term se as well as the value of se at the left ($\mathbf{b.}se$) and right ($\mathbf{e.}se$) end-points of the interval. Using these operators it is possible to express the relationships of a real valued state variable s with its derivative $\dot{s}$ using an integral equation.

$$\mathbf{e.}s - \mathbf{b.}s = \int \dot{s}$$

In the following we will defined EDC as an extension of DC by generalizing the semantic domains, defining new syntax, accompanying semantics and additions to the axioms and rules of the proof system. We exploit the concept of typed states defined in the previous section.

2.4.1 Semantic Domains

When delimiting the class of state functions to use as a model of EDC we need to ensure that the new operators ($\int, \mathbf{b.}, \mathbf{e.}$) can be defined for the class and furthermore that a sufficiently rich collection of operations $\overline{f^n}$ is closed under the chosen class.

Ravn [40] investigates possibilities within these limitations and settles on the general restriction that states must be *piecewise continuous* ([40] definition 2.3). Real valued state are further restricted to the class of *piecewise analytic* ([40] definition 2.4) functions. The continuity of discretely typed state functions imply finite variability.

The properties are preserved by all operations on discrete types, arithmetic operators on reals and inequalities on reals ([40] definition 2.5).

Ravn observes that the class of analytic functions contains the solutions to the linear differential equations often used in control theory.

These consideration lead to a change in the semantic domain of interpretations $\mathbb{I}\mathrm{ntp}$. For an interpretation $\mathcal{I}$ and a state variable s we will require $\mathcal{I}(s) \in \mathbb{T}\mathrm{ime} \stackrel{\mathrm{pc}}{\to} \mathit{Type}(x)$, where $\stackrel{\mathrm{pc}}{\to}$ the function space of piecewise continuous functions. For real valued states variables we also require $\mathcal{I}(s) \in \mathbb{T}\mathrm{ime} \stackrel{\mathrm{pa}}{\to} \mathbb{R}$, where $\stackrel{\mathrm{pa}}{\to}$ the function space of piecewise analytic functions.

2.4.2 Syntax

We extend the syntax of terms with the new constructs. State terms of type $\mathbb{R}$ are called *state expressions* and denoted se.

$$t \; ::= \; x \mid f^n(t_1, \ldots, t_n) \mid \textstyle\int P \mid \int se \mid \mathbf{b.}se \mid \mathbf{e.}se$$

2.4.3 Type System

The type of the new constructs is $\mathbb{R}$:

$$\begin{array}{lcl} \mathcal{T}(\int se) & = & \mathbb{R} \\ \mathcal{T}(\mathbf{b.}se) & = & \mathbb{R} \\ \mathcal{T}(\mathbf{e.}se) & = & \mathbb{R} \end{array}$$

2.4.4 Semantics

The meaning of terms is defined by extending the definition of $[\![t]\!]$ to cover the new constructs. The meaning of $\int se$ is the integral of se over the interval while the meaning of $\mathbf{b.}se$ and $\mathbf{e.}se$ is the limit of se at the left and right end-point of the interval, respectively.

$$\begin{array}{lcl} \mathcal{I}[\![\int se]\!](\mathcal{V}, [b, e]) & = & \int_b^e \mathcal{I}[\![se]\!](\mathcal{V}, t)\, dt \\ \mathcal{I}[\![\mathbf{b.}se]\!](\mathcal{V}, [b, e]) & = & \mathcal{I}[\![se]\!](\mathcal{V}, b+) \\ \mathcal{I}[\![\mathbf{e.}se]\!](\mathcal{V}, [b, e]) & = & \mathcal{I}[\![se]\!](\mathcal{V}, e-) \end{array}$$

2.4.5 Proof System

EDC extends the proof system with one general rule to interface with results from mathematical analysis. The notation $\vdash_{Mat} F$ is used to express that the formula F is a theorem in mathematical analysis.

$$\frac{\vdash_{Mat} \forall [b, e] \in \mathbb{I}\mathrm{ntv} \cdot R(f(b+), f(e-), \int_b^e f(t)dt, e - b)}{\vdash R(\mathbf{b.}f, \mathbf{e.}f, \int f, \ell)} \; \mathtt{analysis}$$

Apart from this rule a few new axioms are defined to relate the new operators with chop. First of all integrals sum over chop:

$$\frac{}{\vdash \int se = v_1 \frown \int se = v_2 \Rightarrow \int se = v_1 + v_2}\ \texttt{integration}$$

Secondly **b.** and **e.** are distribute on left and right subintervals, respectively:

$$\frac{}{\mathbf{b.}se = v \Leftrightarrow (\mathbf{b.}se = v) \frown \mathit{true}}\ \texttt{values\_b}$$

$$\frac{}{\mathbf{e.}se = v \Leftrightarrow \mathit{true} \frown (\mathbf{e.}se = v)}\ \texttt{values\_e}$$

For continuous states continuity is characterized in the axiom:

$$\frac{}{(\mathbf{e.}se = v_1) \frown (\mathbf{b.}se = v_2) \Rightarrow v_1 = v_2}\ \texttt{continuity} \qquad \text{for continuous state expressions } se$$

and the mean value theorem is expressed as:

$$\frac{}{\begin{array}{l}\mathbf{b.}se = v_1 \wedge \mathbf{e.}se = v_2 \wedge v_1 \leq v \leq v_2 \Rightarrow \\ \quad (\mathbf{e.}se = v) \frown \mathit{true}\end{array}}\ \texttt{mean} \qquad \text{for continuous state expressions } se$$

2.5 Standard Forms

A number of standard forms [25, 40] are useful for specifying requirements and design of control applications. A control design consists of an automaton describing how the controller progresses through a number of *phases* together with commitment formulas for each phase describing the system behaviour in the phase.

We will introduce these standard forms in the following and use them in chapter 4 for the design of the Steam Boiler. First, however, we define some derived operators used to express the standard forms and list derived theorems capturing properties of the new operators.

2.5.1 Derived Operators

The first derived operator is the *followed by* operator used to express that a system in a region of its state space described by a formula A will enter a goal state P.

$$A \rightarrow \lceil P \rceil \quad \hat{=} \quad (\forall r \cdot \Box((A \wedge \ell = r) \frown \ell > 0 \Rightarrow \ell = r \frown \lceil P \rceil \frown \mathit{true}))$$

The operator capture the absence of counterexamples to the above behaviour as expressed in the following theorem:

$$\frac{}{\vdash \neg(A \rightarrow \lceil P \rceil) \Leftrightarrow \Diamond(A \frown \lceil \neg P \rceil)} \ \texttt{fb\_neg}$$

and is monotone like implication:

$$\frac{\vdash B \Rightarrow \mathit{true} \frown A \qquad \vdash_{Mat} P \Rightarrow Q}{\vdash (A \rightarrow \lceil P \rceil) \Rightarrow (B \rightarrow \lceil Q \rceil)} \ \texttt{fb\_monotone}$$

The *leads to* operator is derived from followed by:

$$A \xrightarrow{t} \lceil P \rceil \quad \hat{=} \quad (A \wedge \ell = t) \rightarrow \lceil P \rceil$$

and specifies that the system will progress to the goal state when the system has been in the region A for t time units. If the state space region is described by a simple state predicate the leads to operator is completely characterized by the propagation law:

$$\frac{}{\begin{array}{ll} & t > 0 \Rightarrow \\ & (\lceil P \rceil \xrightarrow{t} \lceil Q \rceil \Leftrightarrow \\ \vdash & \\ & (\forall r \cdot \Box \left(\begin{array}{l} (\lceil P \rceil \wedge \ell = r) \frown (\ell > 0) \Rightarrow \\ \quad r < t \vee \\ \quad r = t \wedge (\ell = t) \frown \lceil Q \rceil \frown \mathit{true} \vee \\ \quad r > t \wedge \\ \quad (\ell = t) \frown (\lceil P \wedge Q \rceil \wedge \ell = r - t) \frown \lceil Q \rceil \frown \mathit{true} \end{array} \right))) \end{array}} \ \texttt{lt\_prop}$$

from which the following two theorems can be derived:

$$\frac{}{\vdash t > 0 \Rightarrow (\lceil P \rceil \xrightarrow{t} \lceil \neg P \rceil \Leftrightarrow \Box(\lceil P \rceil \Rightarrow \ell \leq t))} \ \texttt{lt\_bound}$$

$$\frac{}{\vdash t > 0 \Rightarrow (\lceil P \rceil \xrightarrow{t} \lceil Q \rceil) \Rightarrow \Box(\lceil P \rceil \Rightarrow (\ell \leq t \vee (\ell = t) \frown \lceil P \wedge Q \rceil))}\ \texttt{propagation}$$

Timed bounded transitions are expressed by what we will call the *up to* operator which is also derived from leads to.

$$A \overset{\leq t}{\Rightarrow} \lceil P \rceil \quad \widehat{=} \quad (A \wedge \ell \leq t) \rightarrow \lceil P \rceil$$

The up to operator can express unconditional stability of a state P characterized in two ways by the following theorems:

$$\frac{}{\vdash (\lceil \neg P \rceil \frown \lceil P \rceil \overset{\leq t}{\Rightarrow} \lceil P \rceil) \Leftrightarrow \Box(\lceil \neg P \rceil \frown \lceil P \rceil \frown \lceil \neg P \rceil \Rightarrow \ell > t)}\ \texttt{stable}$$

$$\frac{}{\vdash \begin{array}{l} (\lceil \neg P \rceil \frown \lceil P \rceil \overset{\leq t}{\Rightarrow} \lceil P \rceil) \Leftrightarrow \\ \Box \left(\begin{array}{l} \ell > 0 \wedge \ell < t \Rightarrow \\ \lceil P \rceil \vee \lceil \neg P \rceil \vee (\lceil \neg P \rceil \frown \lceil P \rceil) \vee \\ (\lceil P \rceil \frown \lceil \neg P \rceil) \vee (\lceil P \rceil \frown \lceil \neg P \rceil \frown \lceil P \rceil) \end{array} \right) \end{array}}\ \texttt{stable\_2}$$

Conditional stability of P depending of a state Q is partially characterized by this theorem:

$$\frac{}{\vdash \begin{array}{l} (\lceil \neg P \rceil \frown \lceil P \wedge Q \rceil \overset{\leq t}{\Rightarrow} \lceil P \rceil) \Rightarrow \\ \quad \Box(\lceil \neg P \rceil \frown (\lceil P \rceil \frown \mathit{true} \wedge \lceil Q \rceil) \wedge \ell \leq t \Rightarrow \lceil \neg P \rceil \frown \lceil P \rceil) \end{array}}\ \texttt{stability}$$

The above theorems have all been derived using the tool described in chapter 6. The derivation can be found in appendix A.

2.5.2 Standard Forms

The standard forms for controller design and specification are the following: Let $\pi, \pi_1, \ldots$ denote single phase states and let $P, P_1, \ldots$ denote state assertions on the plant or phase states.

Sequencing: $\lceil \pi \rceil \rightarrow \lceil \pi \vee \pi_1 \vee \cdots \vee \pi_n \rceil$ expresses that the successor phase of π is one of the phases $\pi_1, \ldots, \pi_n$.

Progress: $\lceil \pi \wedge P \rceil \xrightarrow{t} \lceil \neg\pi \rceil$ specifies an upper bound t on the time spend in phase π when the condition P holds for the duration.

Selection: $\lceil \neg\pi \rceil \frown \lceil \pi \wedge P \rceil \overset{\leq t}{\rightarrow} \lceil \pi \vee \pi_1 \vee \cdots \vee \pi_n \rceil$ expresses that the sequencing is constrained to enter one of π_1 to π_n by P for t time units. Phase stability is the special case n=0.

Synchronization: $\lceil \pi_1 \vee \cdots \vee \pi_n \rceil \xrightarrow{t} \lceil P \rceil$ expressing that the combined phase $\pi_1 \vee \cdots \vee \pi_n$ will cause P to hold within t time units.

Framing: $\lceil \neg\pi \rceil \frown \lceil \pi \wedge P \rceil \overset{\leq t}{\rightarrow} \lceil P \rceil$ expresses stability of P for t time units when π is entered. This is dual to phase stability.

Critical Duration: $(\int P > d) \overset{\leq t}{\rightarrow} \lceil \neg P \rceil$. This is a form used in requirements specification only and express that d is an upper bound on the duration of P in any interval no longer than t.

For controller design where requirements and design specification is restricted to the above forms the question of satisfaction of the requirements by the design is decidable when t is an integer constant [28].

2.6 Summary

We have presented Interval Logic and its extension Duration Calculus. The starting point was Hilbert style systems and we developed the logics in the Sequent style and generalized them to cover typed states. We will base our further work on the sequent style.

Chapter 3

PC/DC

This chapter describes the Proof Checker for Duration Calculus (PC/DC), a proof tool for DC developed by Skakkebæk and described in his PhD-thesis [45].

We present this work to form a basis for the following chapter which will describe a case study performed using the tool and the experiences gained by the work. It is these experiences of employing PC/DC that have caused us to consider alternative ways to achieve proof support for DC and have ultimately lead to the development of the Isabelle/DC tool described in chapter 6 and onwards.

PC/DC is implemented on top of the Prototype Verification System (PVS) developed by SRI [31]. PVS supplies the facilities for managing specifications expressed in a higher order logic and features a proof tool for this logic.

We will start the chapter by briefly introducing PVS and will then describe details of the PC/DC implementation.

3.1 PVS

PVS is an integrated environment for creating, managing and analyzing formal specifications. Specifications are expressed using a higher order logic with strict typing and are organized into *theories* containing declarations of types, values (and functions), axioms and lemmas.

The Emacs editor [47] provides the user interface for creating, editing and otherwise managing these theories.

Analysis of specifications is done by expressing and proving assertions. Proving is an interactive process where the user builds a proof tree bottom-up from the goal (assertion) to be proven. At each proof step the user applies *strategies*, sequences of atomic proof commands, to the current goal to produce a set of subgoals. The Emacs editor also provides the user interface for the prover.

The strength of the PVS environment lies in the combination of the versatility of the specification language, the comprehensiveness of the library of theories of common datatypes (sets, lists, numbers, etc.) and the power of the prover. Although the proof process is interactive the user may program his own strategies and strong automatic decision procedures are provided for propositional, equational and linear real arithmetic reasoning.

3.2 PC/DC

A prominent part of the implementation of PC/DC is the encoding of DC, the *source* logic, into the logic of PVS, the *base* logic. When encoding, there is a choice of doing a syntactic or a semantic encoding (or indeed both). In a *syntactical* encoding the syntax of the source logic, its proof rules and the concepts of proof and theorem are introduced as entities of the base logic. While for a *semantical* encoding it is the semantical model of the source logic together with the concept of validity which is formulated in the base logic.

PC/DC is based on a purely semantical encoding of DC. As an advantage this allows the axioms and proof rules of DC to be proven sound within the base logic.

The problem of having only a semantical encoding is to present to the user the illusion of doing proofs using the syntax and proof rules of DC. For this purpose PC/DC consists of a set of PVS strategies for applying DC proof rules and a parser/unparser for converting back and forth between the syntax of DC and its semantic encoding.

3.2.1 Encoding

We start our description of the implementation by presenting details from the encoding of DC.

Validity The aim of any semantical encoding is to capture the concept of validity of formulas in the object logic. Recall from section 2.1.2 that validity of a DC formula A is defined from A's meaning $[\![A]\!]$ by quantification over all interpretations, valuations and intervals.

A straight-forward approach to encoding validity would thus be to encode the formula syntax, the semantical domains and the meaning function $[\![A]\!]$ and to express the quantification directly.

PC/DC takes a different approach in order to get a more convenient encoding. First of all the meaning function is not encoded in its entirety instead it is split up into definitions of functions expressing the meaning of individual language constructs ($\neg, \vee, \forall, ^\frown, \ldots$). These functions takes as arguments the meaning of constituent formulas. This approach does away with the need of encoding the formula syntax. Expressing the meaning of a formula is now a question of composing the meaning functions of its subformulas. This composition is a kind of translation.

Secondly the domains of interpretations and valuations are not encoded either. Instead the meaning of state variables and global variables are represented directly as variables of the base logic.

Expressing validity of a formula A in the PC/DC encoding is then done by composing its meaning and quantifying over the PVS variables representing the meaning of state and global variables.

It is not immediate that the approach just described is faithful to the semantics presented in 2.1.2. Section 4.2.2 of [45] contains a discussion of this subject.

Encoding details In the following we add more detail to the description of the encoding and present excerpts of the PVS theories forming the semantical encoding of DC. The encoding takes the form of a number of PVS theories and consists of:

- An encoding of the basic semantical domains of time, interval, state, term and formula.
- An encoding of the semantics of operators and quantifiers.
- An encoding of validity of DC formulas.
- An encoding of the axioms and proof rules of DC.

Domains The semantic domains are encoded directly as PVS types.

```
Time : TYPE = real
b, e : Time;
Interval : TYPE = {(b, e) | b <= e};

f : [ Time -> bool ];
State : TYPE = { f | Step_Function(f) };
Step_Function(f) : bool = ... ;

IReal : TYPE = [ Interval -> real ];
IForm : TYPE = [ Interval -> bool ];
```

`State` is the type capturing the meaning of state assertions and state variables as boolean functions over time which are finitely variable. `Step_Function` express this property of states. State may have types other than boolean. However, the encoding of this is not presented here. `IReal` and `IForm` are the types capturing the meaning of terms and formulas, respectively.

Notice that the semantical concepts of interpretation and valuation are *not* encoded. Instead a DC state variable s is captured directly by its meaning $\mathcal{I}(s)$ represented as PVS functions while a DC rigid variable x is translated into a PVS variable representing $\mathcal{V}(x)$. This way PVS is used to manage the name space.

Operators and quantifiers The encoding of the logical operators come in two versions: A pointwise version for use in state expressions and another for intervals used in formulas. This is illustrated below for the `\/` operator:

```
i : Interval;
s1, s2 : State;
A, B, C, D : IForm;
\/(s1, s2) : State = (LAMBDA (t:Time) : s1(t) OR s2(t));
\/(A, B)(i) : bool = A(i) OR B(i);
```

`OR` is the built-in PVS operator for disjunction. The same principle of lifting is used for the arithmetic operators on terms (`+ - * /`), relations (`= <=`...) and constants (`n`).

The duration operator (`dur`) is declared as an uninterpreted function and its properties are axiomatized:

```
NonNegReal : NONEMPTY_TYPE = {x:real|x>=0} CONTAINING 1
INonNegReal : TYPE = [Interval -> NonNegReal];

p, q : VAR State

dur(p) : INonNegReal

axiom1: AXIOM dur(ff) = n(0)

axiom2: AXIOM valid(dur(p)>=n(0))

axiom3: AXIOM dur(p)+dur(q)=dur(p/\q)+dur(p\/q)

r,s : VAR NonNegReal

axiom4: AXIOM (dur(p)=n(r))^(dur(p)=n(s)) = (dur(p)= n(r)+n(s))
```

State predicates and points are abbreviations:

```
p : VAR State

pr(p) : IForm = dur(p)=len /\ len>=n(0)
point : IForm = len=n(0)
```

The chop operator is encoded as `^`:

```
b : Time = proj_1(i);
e : Time = proj_2(i);
^(A, B)(i) : bool =
  (EXISTS (m:Time) : b <= m AND m <= e AND A(b, m) AND B(m, e));
```

I.e. there exists some time point `m` between the end-points of the interval `i` such that the formula `A` is true for the left subinterval and the formula `B` is true for the right subinterval. A straight-forward encoding of the semantics of chop.

Quantification is encoded using the PVS concept of a parametric theory or theory schema. Theory schemas have formal parameters with may be types or values. To form theories theory schemas are instantiated with actual parameters.

The theory schema encoding DC quantification has a formal type parameter `t`. `t` is the type to quantify over.

```
fall(A : [t -> IForm])(i) : bool = (FORALL (r : t) : A(r)(i));
ex(A : [t -> IForm])(i) : bool = (EXISTS (r : t) : A(r)(i));
```

To see how this works consider the encoding of the DC formula

$$(\exists x \cdot \ell = x)$$

The theory schema is instantiated replacing `t` with `real` to introduce the declaration

```
ex(A : [real -> IForm])(i) : bool = (EXISTS (r : real) : A(r)(i));
```

and the formula is encoded as

```
ex(LAMBDA (x : real) : len > n(x))
```

which evaluates to:

```
(EXISTS (r : real): (LAMBDA (x : real) : len = n(x))(r))
```

and further to:

```
(EXISTS (r : real): len = n(r))
```

where `LAMBDA` is the PVS operator for lambda abstraction.

Validity Proving a formula in PC/DC amounts to establishing the validity of the formula. For a formula `A` the goal have the form `valid(A)` where `valid` encodes validity:

```
valid(A) : bool = (FORALL (i:Interval): A(i))
```

Quantification is only over intervals as the quantification over interpretations and valuations is replaced by implicit quantification over PVS variables representing the meaning of state and global variables. As an example recall the axiom $L1$:

$$(\ell = x) \frown A \Rightarrow \neg((\ell = x) \frown \neg A)$$

Validity of the axiom $L1$ can be formulated as:

```
L1 : THEOREM valid((len=n(x))^A => NOT((len=n(x))^(NOT A)))
```

where `x` and `A` are variables representing the meaning of x and A.

All PVS theorems are closed so the universal quantification `x` and `A` is implicit when stating theorems. During proof the quantification becomes explicit in the goal:

```
(FORALL (x:real): (FORALL (A:IForm):
   valid((len=n(x))^A => NOT((len=n(x))^(NOT A)))))
```

Proof system For the implementation a Gentzen style proof system was developed for DC and validity of a sequent expressed from the encodings of the constituent formulas:

```
Gamma, Delta : IForm;
seq(Gamma, Delta) : bool = (FORALL i : (Gamma(i) IMPLIES Delta(i)));
```

Antecedent and consequent are encoded in the same domain (`IForm`) as formulas. The antecedent is encoded as the conjunction of the encodings of the constituent formulas and the consequent as the disjunction. When formulating rules, operators `*` and `+` represent conjunction and disjunction, respectively.

```
Gamma, Gamma' : IForm;
*(Gamma,Gamma')(i) : bool = (Gamma(i) AND Gamma'(i));

Delta, Delta' : IForm;
+(Delta,Delta')(i) : bool = (Delta(i) OR Delta'(i));
```

Proof rules are formed from a pair of premises and a conclusion, all sequents. Rules are sound i.e., they preserve validity of sequents:

```
Prem1, Prem2, Concl : bool;
rule(Prem1, Prem2, Concl) : bool = Prem1 AND Prem2 IMPLIES Concl;
```

An example is the encoding of a rule for the monotonicity of chop:

```
mono: LEMMA rule(
                seq(Gamma,[](A=>C) + Delta),
                seq(Gamma,[](B=>D) + Delta),
              %------------------------------
                seq(Gamma * A^B, C^D + Delta))
```

Rules are declared as PVS lemmas and proven, this establish the soundness of the rules.

Structure and size The encoding is structured as 20 PVS theories and comprise some 1400 lines of specification. Figure 3.1 show the import relationship of the theories. A line in the figure means that the upper theory directly imports properties from the connected lower theory. Collectively the specification contains some 240 proofs to demonstrate the soundness of encoded DC rules. These proofs comprise a total of just over 3000 lines.

3.3 EDC Extension

For the purpose of modeling the Steam Boiler introduced in the next chapter we make a small extension to PC/DC adding the operators of EDC (see section 2.4) and the type of continuous real-valued states.

`ContState` as a subtype of the functions of from time to real intended to model continuous functions.

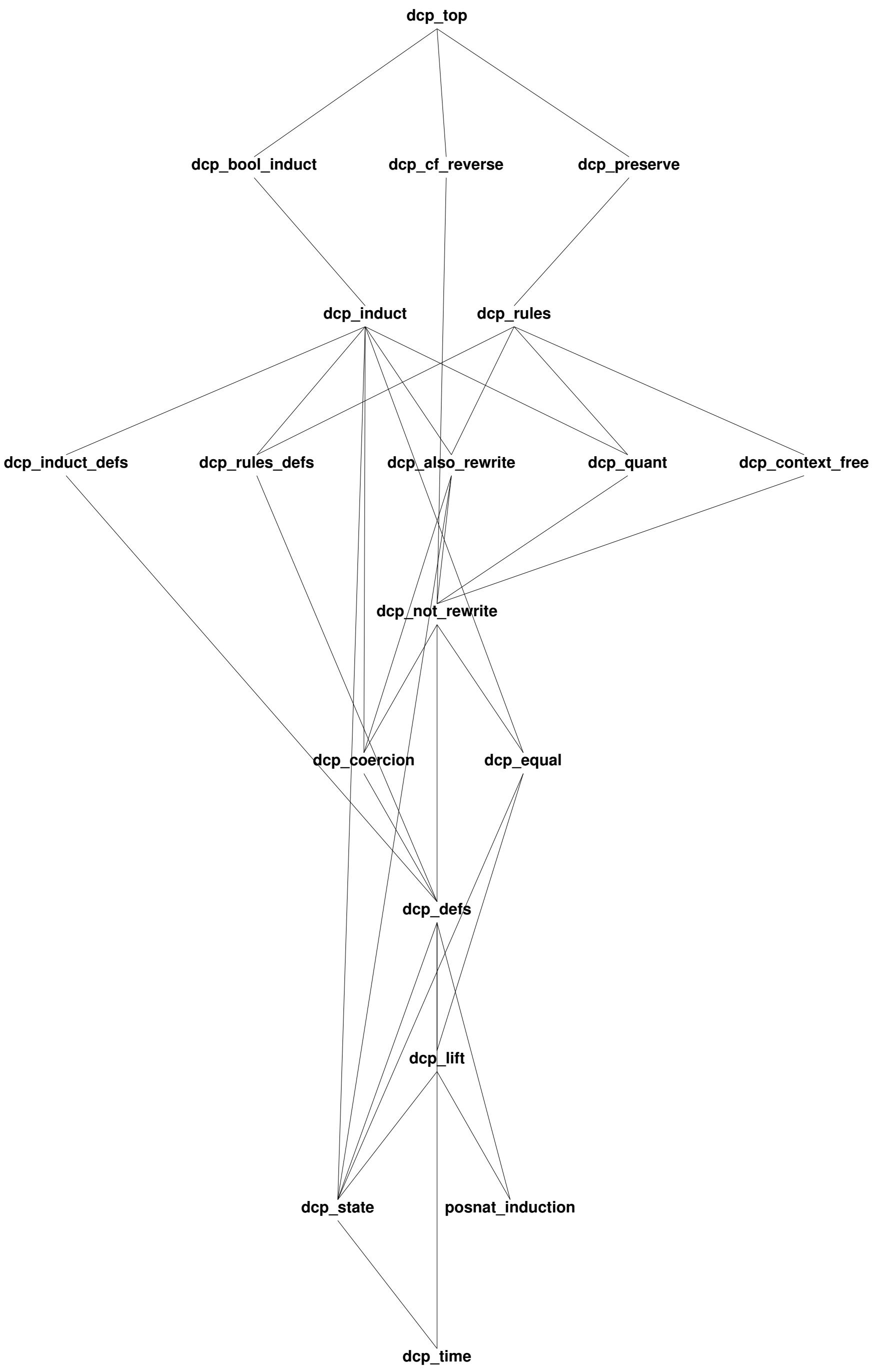


Figure 3.1: Theory hierarchy of DC encoding

```
ContState : NONEMPTY_TYPE From [Time -> real]
```

We are quite vague about this type but we do require that pointwise comparison of two `ContState`s yield a finitely variable boolean state and that lifting a constant yields a continuous state. This is axiomatized as follows:

```
  equiv_finvar: AXIOM
      Finite_Variability[bool](LAMBDA t: cs1(t) = cs2(t));
  greaterp_finvar:
      AXIOM Finite_Variability[bool](LAMBDA t: cs1(t) > cs2(t));
  c_contstate: AXIOM ContState_pred(LAMBDA t: r);
```

We can now equip the state type with useful relations via pointwise extension:

```
cs1, cs2 : VAR ContState
t : VAR Time

==(cs1, cs2) : State = LAMBDA t: cs1(t)=cs2(t);
>(cs1, cs2) : State = LAMBDA t: cs1(t)>cs2(t);
c(r) : ContState = LAMBDA t: r;

<(cs1, cs2) : State = NOT (cs1==cs2 \/ cs1>cs2);
<=(cs1, cs2) : State = NOT cs1>cs2;
>=(cs1, cs2) : State = cs1==cs2 \/ cs1>cs2;
```

The special EDC operators $\int$, **b.**, **e.** are declare by:

```
b, e : [ContState -> [Interval -> real]]
intg : [ContState -> [Interval -> real]]
```

and axiomatized:

```
cs : VAR ContState
v, v1, v2 : VAR real

b_value : AXIOM (b(cs)=n(v)) = ((b(cs)=n(v))^TRUE)
b_chop : AXIOM (b(cs)=n(r) /\ A^B) = ((b(cs)=n(r) /\ A)^B)
b_intro : AXIOM A^B = (ex! r: (b(cs)=n(r) /\ A)^B)
```

Properties of continuous states are axiomatized as DC axioms:

```
Continuity : AXIOM (e(cs)=n(v1));(b(cs)=n(v2)) => n(v1)=n(v2)
cont_le : AXIOM 'cs<=c(v)' => b(cs)<=n(v) /\ e(cs)<=n(v)
Mean_lt : AXIOM len>n(0) /\ (b(cs)=n(v1)) /\ (e(cs)=n(v2)) /\
                n(v1)>=n(v) /\ n(v)>n(v2) =>
                    TRUE;(b(cs)=n(v) /\ 'cs<c(v)')
```

3.4 Standard Forms Extension

The standard forms from section 2.5 are also added to PC/DC as an extension to be used in the Steam Boiler example. The definitions of these derived forms is straight-forward:

```
A, B : VAR IForm
r, t : VAR real

fby(A, B) : IForm =
  (fall! r: [](A /\ (len=n(r)) ^ (len>n(0)) => (len=n(r)) ^ B ^ TRUE))

lto(A, t, B) : IForm = fby(A /\ (len=n(t)), B)

upto(A, t, B) : IForm = fby(A /\ (len<=n(t)), B)
```

To allow for a slightly more convenient syntax the DC parser is augmented with infix forms of these definitions:

Operator	PVS definition	ASCII syntax
$A \rightarrow B$	`fby(A, B)`	`A --> B`
$A \xrightarrow{t} B$	`lto(A, t, B)`	`A -:= t :-> B`
$A \xrightarrow{\leq t} B$	`uto(A, t, B)`	`A -:<= t :-> B`

3.5 Summary

We have presented PC/DC the DC proof tool by Skakkebæk. We provided excerpts of the semantic encoding of DC on which the tool is based and described a few extensions we have made to the tool prior to applying it on the Steam Boiler case study.

Chapter 4

Steam Boiler Case Study

In this chapter we will present a case study of the use of PC/DC. Experience gained from this and other work has lead us to consider other ways of obtaining mechanized proof support for DC. We summarize the experiences at the end of the chapter, but first we introduce the case study of the Steam Boiler and how it was captured in PC/DC. The material in this chapter is to a large extent taken from [41]. In Chapter 9 we revisit the case study to redo the verification using Isabelle/DC.

4.1 Background

The Steam Boiler was proposed by Abrial, Börger and Langmaack [2] as a challenge to the large variety of formal specification approaches developed today. Researchers would specify the problem using a technique of their choice and the solutions would be compared at the Dagstuhl workshop in 1995. The multitude of entries into this contest are documented in [1].

The particular specification of the problem chosen for this case study is one by Rischel et al. [42]. The approach focuses on the safety properties of the system and develops a control design using a mix of DC and TLT (Temporal Language of Transitions) [12] specifications. The paper also includes pen and paper proofs of main theorems and lemmas.

With the specification of the problem given by [42] the aim of this case study is to redo and mechanize the proofs of [42] using the PC/DC tool.

4.1.1 TLT

TLT developed by Cuéllar [12] is basically a notation for specifying a transition system consisting of a state space, an initial condition and a next-step relation.

The state space is defined by declaring state variables. The initial condition is a predicate over these variables. The next state relation is defined via transitions over the present state variables and the next state variables (primed).

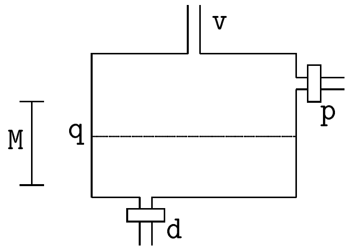
The subset of TLT we will be using specifies the transitions of the system in the form $lhs_i \Rightarrow rhs_i$ for $i \in \{1..n\}$ where *lhs* and *rhs* are predicates over the state variables (present and next). If a next state variable v' occurs in a right hand side predicate (rhs_i) it is *controlled* by transition i. The resulting predicate defining the next state relation is a conjunction. Each transition contributes with a conjunct $lhs_i \Rightarrow rhs_i$. A controlled variable v' further contributes with a conjunct being a framing constraint of the form $v \neq v' \Rightarrow (lhs_{j_1} \vee \cdots \vee lhs_{j_k})$ where $\{j_1, \ldots, j_k\}$ are the transitions which control v'.

4.1.2 Related Work

As mentioned above, [1] contains a number of solutions to the Steam Boiler problem. These can be categorized according to characteristic features: Is the specification language algebraic, logical or operational? Does the modeling of the system and environment involve continuous behavior or is it purely discrete? Does the solution involve proof formal or not? A subdivision of the solutions based on logic is according to the kind of logic used: Temporal, real-time, higher order.

Two approaches resemble the one followed here: In [43] there is a solution based on a continuous system model and requirements specification in DC. It contains a functional design with transformations to an Occam-like program. System properties are proven by hand proofs. In [49] is a specification based on an real time extension of Hoare program logic encoded in the higher order logic PVS. The system model is continuous. System properties are proven in PVS. This approach adapts the semantic encodings to the problem at hand. It uses PVS to its best but with much freedom.

4.2 Steam Boiler



The problem of the steam boiler is to device a control system to keep the water level `q` inside the tank of a steam boiler within a given interval `M` once the system is started. The control of the heater for actually boiling the water is not part of the problem.

The boiler is equipped with a valve to drain excess water, more water may be added to the boiler through the pump and the produced steam escapes through a pipe in the top of the boiler.

`d`, and `p` are the throughputs of water through the valve and pump, respectively. `v` is the throughput of steam through the escape pipe.

The control of the water level is done in two phases. In the initial phase the water level is brought within a given range using the pump and the valve after which the control enters the `ready` phase where boiling can commence and the controller adds water to the tank as needed through control of the pump.

4.2.1 The Plant

From the perspective of the control system the water level q and the throughputs p, q and v belong to the environment. The variables vary continuously with time and the link between them are conveniently expressed by the equation:

```
Mass : AXIOM e(q) - b(q) = intg(p) - intg(v) - intg(d)
```

The equation simply states that over a time interval the change in water level (the different between end value and the beginning value) is given by the integrals of the throughputs over that interval.

4.2.2 The Commitments

The control system shall satisfy the following five commitments. `Init` is a formula describing the proper initialization of the control system.

```
Commit1 : THEOREM Init => point \/ 'NOT ready';TRUE
Commit2 : THEOREM Init => 'ready'-->'ready'
Commit4 : THEOREM Init => 'ready' --> 'd=0'
Commit5 : THEOREM Init => 'ready' --> 'member(q, M)'
```

The system shall start in the initial (not `ready`) phase (`Commit1`) and once entered, the `ready` phase is never left (`Commit2`). The system shall keep the valve closed while in the `ready` phase (`Commit4`) and finally during the `ready` phase the water level `q` shall be kept within the interval `M` (`Commit5`).

4.2.3 The Controller Design

The architecture of the control system consists of four components in a very traditional layout. A sensor to measure the water level, two actuators for controlling the pump and the valve and of course the control program.

The water level is sampled at regular intervals by the water level sensor. The sensor is allowed some inaccuracy regarding the measured water level. The value available to the control program as the variable `adj` is thus a water level interval but the sensor is to guarantee that the actual water level is contained in this interval.

The actuators controlling the pump and value are controlled by two boolean program variables `pCmd` and `vCmd`, respectively. When true the pump and valve are commanded to open, when false to close. Opening and closing happens with a delay. Only after time `tau` is the equipment required to respond. This is specified as follows for the pump (and analogous for the valve):

```
PA1 : AXIOM 'pCmd' -: tau :-> 'p=P'
PA2 : AXIOM 'NOT pCmd' -: tau :-> 'p=0'
```

That is the pump reaches full throughput (`P`) `tau` time units after the program commands it to open and conversely closes in `tau` time units.

The control program is written in TLT and basically specifies the initial values `I` of the program variables and a transition relation `R` between pre and post values of the variables. A computation is then simply an iteration of the transition relation starting from the initial state.

The computation evolves over time by one step (one iteration of the transition relation) each `tau` time unit. This is specified by synchronizing the computation steps with an integer clock `a`. `a` increases by one each `tau` time unit and is kept stable within each period:

```
AC1 : AXIOM 'a=n' --> 'a=n \/ a=n+1'
AC2 : AXIOM 'a=n' -: tau :-> 'NOT a=n'
AC3 : AXIOM 'NOT a=n';'a=n' -:<= tau :-> 'a=n'
AC4 : IForm = point \/ ('a=0' /\ len < tau) \/
                (('a=0' /\ len = tau);TRUE)
```

Notice the ASCII syntax for the *leads to* and *up to* operators introduced in 2.5. `AC4` ensures that initially `a` is zero and remain so for the first period. `AC4` is a conjunct of the `Init` formula mentioned above.

The synchronization of the computation with the clock is expressed by stating that the collective program state `s` can only change when `a` changes.

```
Sync_s : LEMMA 'a=n /\ s=s0' --> 'NOT a=n \/ s=s0'
```

And furthermore that the program state is properly initialized and changes only according to the transition relation `R`.

```
Sem1 : IForm = point \/ 'I(s)';TRUE
Sem2 : AXIOM 's=s1';'s=s2' => R(s1, s2)
```

`Sem1` is the other conjunct of the `Init` formula.

```
Init : IForm = AC4 /\ Sem1
```

Having `Init` as a precondition on all the commitments expresses that the control system only commits to the requirements if properly initialized.

Variables
  Read $\#a$: Nat INIT 0
  Read $adj(\mathbf{1})$: Interval INIT $[0,C]$
  Private pRd: Bool INIT 0
  Private $pCmd, vCmd$: Bool INIT 0
Always (* *Control* *)

$$\begin{array}{l}
[\!] \ \#a\uparrow \ \wedge \neg pRd \ \wedge \neg vCmd \ \wedge \neg pCmd \ \wedge (adj'(\mathbf{1}) \succ N) \ \Rrightarrow \ vCmd' \\
[\!] \ \#a\uparrow \ \wedge \neg pRd \ \wedge (adj'(\mathbf{1}) \subseteq NN) \ \Rrightarrow \ \neg vCmd' \ \wedge \ \neg pCmd' \\
[\!] \ \#a\uparrow \ \wedge \neg pRd \ \wedge \ \neg vCmd \ \wedge \neg pCmd \ \wedge (adj'(\mathbf{1}) \subseteq N) \ \Rrightarrow \ pRd' \\
[\!] \ \#a\uparrow \ \wedge \neg vCmd \ \wedge \neg pCmd \ \wedge (adj'(\mathbf{1}) \prec N) \ \Rrightarrow \ pCmd' \\
[\!] \ \#a\uparrow \ \wedge pRd \ \wedge (adj'(\mathbf{1}) \succ N) \ \Rrightarrow \ \neg pCmd'
\end{array}$$

The above is the TLT program for the controller. In the initial phase the program controls the valve and the pump with the aim of getting the water level within the interval NN. When this is done the program goes to the `ready` phase where the goal is to keep the water level within the interval M. Whenever the water level rises above the interval N the pump is stopped and if it drains below N the pump is started to add water.

The program is specified directly as the initialization predicate and the relation on pre state `s` and post state `sp` denoted by the TLT program as explained in section 4.1.1.

```
I(s0) : bool = ...
R(s, sp) : bool = ...
```

Properties of `R` are formulated as PVS lemmas. One is:

```
TLTLem21 : LEMMA R(s1, s2) AND pRd(s1) IMPLIES pRd(s2)
```

saying that once the controller is in the `ready` phase, it stays there.

Reading of the water level sensor is also synchronized with incrementing the clock. The following furthermore says that the sample is correct in that the actual water level is contained in the water level interval output to the program through `adj`.

```
WLS1 : AXIOM (e(q)=q0 /\ 'a=n') --> 'a=n \/ member(q0, adj)'
```

Basically this is the design. Much of course depends on `R` (the program) and its properties but from these few formulas and axioms it is possible to prove the commitments.

4.3 Verification

Most commitments and lemmas are followed by formulas `D --> 'P'`. Hand proofs of such theorems are usually by contradiction using a chained implicative style: Assuming the formula is violated we derive false. The proof of commitment 2 is an example:

$$
\begin{array}{ll}
\neg(\lceil ready \rceil \rightarrow \lceil ready \rceil) & \Rightarrow \quad \{\rightarrow \text{expansion}\} \\
\Diamond\,(\lceil ready \rceil\,;\lceil \neg\, ready \rceil) & \Rightarrow \quad \{\,\Box\,(\lceil ready \rceil \iff \lceil \mathtt{pRd} \rceil)\} \\
true\,;\lceil \mathtt{pRd} \rceil\,;\lceil \neg\ \mathtt{pRd} \rceil\,;true & \Rightarrow \quad \{FinVar\} \\
true\,;\lceil \mathtt{pRd} \wedge \mathtt{s} = s_1 \rceil\,;\lceil \neg\ \mathtt{pRd} \wedge \mathtt{s} = s_2 \rceil\,;true & \Rightarrow \quad \{Sem_2\} \\
R(s_1, s_2) \wedge pRd(s_1) \wedge pRd(s_2) & \Rightarrow \quad \{TLTLem_{2,1}\} \\
pRd(s_2) \wedge \neg\ pRd(s_2) & \Rightarrow \quad \{\} \\
false &
\end{array}
$$

The proofs in PC/DC follow a similar style but are direct, not by contradiction, proving the contrapositive of each implicative step of the hand proof.

A proof is broken down into a series of lemmas, one for each proof step and ending with the theorem to prove. Each lemma is then established using the previous lemma in the chain.

Using this approach gives a lot of lemmas but has the advantages that it allows for proof planning and that the idea of the proof can be inspected in the specification and is not buried internally in the tool. Furthermore proof steps in the formalized proof are finer grained than in hand proofs, the subdivision helps to keep individual proofs smaller and more focused.

4.4 Metrics

In this section we will present some proof metrics, this is, quantitative measurements reflecting the size of the proofs and the frequency individual proof commands are used in the proofs. These numbers will point the way to areas where PC/DC can become a better tool and minimize the manual labor involved in the proof process.

How to Measure The basis for the analysis is the `.prf` files that PVS stores in the context directory alongside the `.pvs` theory files and various derived information.

For the presentation here it is enough to view the `.prf` files as nothing more than a mapping from theorem or lemma names to proofs scripts. Proof scripts are the series of proof commands the user issues while doing a proof in PVS. The structure of the proof scripts reflect closely the proof tree built in PVS. In a proof script it is possible to identify each proof command, its name, parameters etc. Proof files and proofs scripts are manipulated by LISP code built for this particular purpose.

What to Measure What is interesting to know for a case study like the Steam Boiler is its size. How many proof command where necessary in the proofs? While this number does not say anything conclusive about the complexity of the proofs, it *will* say something about the amount of manual labor put into the task. The size in terms of issued proof commands is easily measurable.

To get closer at a characterization it is also interesting to measure the frequency with which a particular proof command is used. This is interesting overall but also as a way to compare proofs of individual theorems and lemmas. Again these numbers are easy to get at.

As is usually the case, it is far easier to generate metrics than to analyze them for information. The proofs of the Steam Boiler uses more than 200 different proof commands. So there is a need to group these into a manageable number of groups to reduce the data material.

The groups are enumerated in the table of figure 4.4 and collectively contain more the 90% of the proof commands.

Time is another important measure. What is the total running time of all proofs? Proving in PVS is an interactive process. Proofs are rerun, augmented, altered and discarded. Theories are developed, expanded, type checked and re-proven. The time take for these step is important for the usability of the tool. Notice that increasing the level of automatic reasoning by PVS and decreasing the number of user entered proofs commands may not reduce the actual (re-)running time of proofs. The measurements dealt with here are of user entered proof commands. PVS may expand these to a far larger number of primitive proof commands.

4.5 Discussion

What can we conclude from the numbers in the Table. Here is a list of notable observations:

- The *struct* group accounts for around 8% of the commands used. A closer analysis reveals that the use of `hide` is the major contributor to this large number. Hiding unused formulas makes the sequent smaller and also helps to narrow the search space for various DC strategies.

  The large number of unused formulas is to some extent a result of the proof style with forward reasoning in the antecedent. This is the case when using monotonicity commands and case splitting. One might consider that these proof commands as default remove the source antecedent formulas leaving only the result.

- The *prop* group accounts for around 18% of the commands used. A closer analysis reveals that the use of `ground` and `assert` is the major contributors to this number.

  `ground` and `assert` require no user supplied parameters. It is tempting as part of the other DC proof commands to do a `ground` just in case it might help. This would free the user of issuing the commands but of course this is gained only at the cost of additional and potentially wasted run time.

- Not surprisingly monotonicity of chop is an important and much used property. The *mono* group accounts for 14% of the commands used. The commands in this group generally require user supplied parameters and are instrumental in directing the proof. Automating these commands will mean proof search.

Group id	Description	Count	%
prop	PVS propositional rules	569	17.1
struct	PVS structural rules	263	7.9
equal	PVS equality rules	180	5.4
quant	PVS quantifier rules	140	4.2
rewrite	PVS rewrite and auto-rewrite rules	85	2.6
mono	DC monotonicity of chop	481	14.4
decide	DC decision procedure call	262	7.9
assoc	DC associativity of chop	227	6.8
modal	DC modality rules for box and diamond	174	5.2
dccase	DC case split	121	3.6
false	DC FALSE propagation and chop	86	2.6
dcquant	DC quantifier rules	72	2.2
propdur	DC duration properties	72	2.2
free	DC Context free terms and chop	32	1.0
edc	EDC rules for b. and e.	26	0.8
disj	DC disjunction and chop	9	0.3
sblem	Steam Boiler lemmas	126	3.8
sbax	Steam Boiler axioms and definitions	46	1.4
math	Lemma about continuous states	28	0.8
other	Ungrouped commands	330	9.9
total		3329	100.0

Figure 4.1: Proof command groups.

- Associativity of chop account for 7% of commands used. These commands are usually used to maneuver a formula into the proper form to be used for monotonicity. It is possible to combine the associativity commands with the use of monotonicity. This has been done in a more recent version of PC/DC. However this new feature was not available for the major part of the Steam Boiler proofs.
- The *propdur* group contains the `p-monotone` command. This command links DC reasoning with mathematical reasoning.
- The commands in the *math* group are mostly lemma introductions of properties of continuous states. These properties are clearly necessary in this case study, but still they are used in less the 1% of the proof steps.

A detailed study of selected proofs show that improved combination of monotonicity and associativity, automatic hiding of formulas and selective use of the decision procedures `ground` and `assert` may reduce the command count by as much as 50%.

This study has not touched upon automatic proof search. The work presented in the following chapter involves development and implementation of proof search procedures for subsets of DC. This work is done using the generic theorem prover Isabelle [33]. Isabelle is found to provide a more open and well documented development platform than PVS for work of this kind. The experiences of the present study carries over to this work.

Chapter 5

Isabelle

Isabelle is a *generic* system for building proof tools for a wide variety of logics. With Isabelle, the possible level of automated reasoning support span the complete range from mere proof editors with no automated deduction to complete theorem provers or decision procedures. Often Isabelle is used to make semi-automatic *proof assistants*. Assistants have been developed not only for classical logics like first order logic, Zermelo-Fraenkel set theory, higher order logic and first order sequent calculus [35] but also for more specialized logics like the logic of partial functions [3], advanced program logics like VTLoE [16] and relational theories like Ruby[39]. Here we will use Isabelle to construct a proof assistant for DC.

The basic Isabelle system is itself a proof assistant for an intuitionistic higher order logic known as the *meta-logic*. Instantiating the generic system to support other *object* logics consists in extending this meta-logic with new constants and axioms representing the syntax and the rules for the object logic.

In the following we will introduce central concepts of Isabelle needed in the construction of the proof assistant for DC found in chapter 6.

5.1 Types and Terms

The first step in encoding an object logic is to declare types representing the basic syntactic categories of the logic like formulas and terms. Term constants representing the abstract syntax constructors can then be declared as functions over these types. Isabelle employs a polymorphic type system for the declaration of types and constants.

Types Types or type constants are nullary type constructors (*con*), which in general are declared by stating their name and the number of arguments they take. Examples are the type constant `prop` of meta-logic terms and the function type constructor `fun` declared in pure Isabelle:

```
prop

('a 'b) fun
```

('a 'b) is a list of type variables. The length of the list declare the number of arguments of the type constructor. Thus fun takes two arguments.

Types t are constructed from type constants and type variables through application of type constructors $(t_1, \ldots, t_n)\,con$. Function types may also be written $[t_1 \texttt{=>} t_2]$ with the curried form $[t_1, \ldots, t_n \texttt{=>} t]$

Terms Term constants are declared by stating their name and type. An example is the declaration of the implication operator of the Isabelle meta-logic:

```
"==>" :: [prop, prop] => prop
```

Term constants may be polymorphic using type variables $'a$ in their type declarations like for example the term constant for universal quantification in the meta-logic:

```
"!!"  :: ['a:logic => prop] => prop
```

This example also illustrates the higher order syntax [37] used in Isabelle to encode quantification. !! is declared as an operator on functions from the type 'a quantified over to formulas of the meta-logic with type prop.

Classes The concept of *class* is used to control over which types a type variable ranges. In the example above the type variable 'a ranges over the types in the type class logic.

A *class* (C) is a set of types and classes are ordered in a hierarchy based on subset inclusion. A class is declared by stating its name and its immediate *super-classes* $C < C_1, \ldots, C_n$. Transitive subset inclusions are implicit.

The pure (ie. unextended) Isabelle system declares the class logic. To reason about object logic terms their type must be included in logic.

Sorts A *sort* (S) is a finite set of classes $\{C_1, \ldots, C_n\}$ denoting the intersection of its members $C_1 \cap \cdots \cap C_n$. $\{\}$ is the *universal* sort denoting all types. A singleton sort $\{C\}$ may be written as just C without the braces.

One example of the use of classes and sorts to control polymorphism was the declaration of meta-quantification !! above. A consequence of this declaration is that meta-quantification only is possible on logical types, ie. types in the singleton sort logic.

Meta-equality on the other hand allows terms of any type to be compared as stated in the declaration:

```
"=="  :: ['a:{}, 'a] => prop
```

where {} as the universal sort denotes any type.

Recall that classes are collections of types. Types are inserted into classes by *arity* declarations $con::(S_1,\ldots,S_n)\,C$. Here n must equal the number of arguments of *con*. Such a declaration includes in the class C all types constructed by *con* from types in the sorts S_1 to S_n. The arities for `prop` and `fun` are:

```
prop :: logic

fun  :: (logic, logic)logic
```

so included in the class `logic` are all terms of the meta-logic and all functions between types in that class.

To illustrate the concepts introduced above consider the following example.

Example The declaration of the Minimal Logic of Implication as an object logic could look something like:

```
Hilbert = Pure +

types
  o

arities
  o :: logic

consts
  Trueprop :: o => prop

  "--> "   :: [o, o] => o
```

We introduce the type `o` denoting the set of minimal logic truth-values and declare implication (`-->`) as the only term constant. `Trueprop` is a coercion function or a meta-level predicate on object level truth-values. `Trueprop(`A`)` is interpreted as A *is true in the object logic.*
□

5.2 Meta-Logic

Supported logics are represented in a fragment of higher order logic usually using a syntactical encoding.

The higher order logic, referred to as the *meta logic* $\mathcal{M}$, is intuitionistic and the primitive rules of the proof system is in the natural deduction style. The deduction approach is based on a combination of higher order unification and resolution. The syntax of the meta logic is simple allowing only the implication connective, equality and universal quantification. The syntax of object logics is the simply typed λ-calculus.

5.2.1 Pure Syntax

The unextended or *pure* Isabelle syntax has basically two levels: *prop* and *logic*. *prop* is the syntax of the meta logic while *logic* is the syntax of the object logic[1]. *aprop* are atomic propositions. The syntax of the object logic is the λ-calculus. Ignoring the syntax of types, type constraints and a few other details the Isabelle syntax can summarized by the following grammar[2]:

$$
\begin{array}{rcllr}
any & = & prop \mid logic & & \\
\\
prop & = & \texttt{(}prop\texttt{)} & & \\
 & \mid & \texttt{PROP}\,aprop & & \text{atomic proposition} \\
 & \mid & any^{(3)}\ \texttt{==}\ any^{(2)} & (2) & \text{meta equality} \\
 & \mid & prop^{(2)}\ \texttt{==>}\ prop^{(1)} & (1) & \text{meta implication} \\
 & \mid & \texttt{[|}prop\texttt{;}\ldots\texttt{;}prop\texttt{|]}\ \texttt{==>}\ prop & (1) & \text{chained meta implication} \\
 & \mid & \texttt{!!}id\,\texttt{.}\,prop & (0) & \text{universal meta quantification} \\
\\
aprop & \mid & id \mid var \mid logic^{(\infty)}\texttt{(}any\texttt{,}\ldots\texttt{,}any\texttt{)} & & \\
\\
logic & = & \texttt{(}logic\texttt{)} & & \\
 & \mid & id \mid var \mid & & \\
 & \mid & logic^{(\infty)}\texttt{(}any\texttt{,}\ldots\texttt{,}any\texttt{)} & & \text{application} \\
 & \mid & \texttt{\%}id\,\texttt{.}\,any^{(3)} & (3) & \text{abstraction}
\end{array}
$$

The lexical grammar define two kinds of names: identifiers (*id*) and unknowns (*var*). Unknowns are prefixed with a `?` and are free variables instantiable during proof.

The above grammar is a special kind of context free grammar called a *priority grammar*. This kind of grammar is an convenient way to specify priority and associativity of operators in a context free language. A priority grammar may be expanded into a conventional context free grammar at the expense of introducing a large number of new non-terminals.

Each non-terminal in the productions of a priority grammar is equipped with an integer priority in the range 0 to some (large) number denoted by ∞. Nonterminals (A) on the right-hand side are decorated with their priority p ($A^{(p)}$) while the priority of the left-hand non-terminal appear in the column on the right. Right-hand priorities of 0 and left-hand priorities of ∞ may be omitted.

The priorities influence the way derivations in the grammar are made in the following way: Let α, β and γ be strings of terminal or non-terminal symbols. Then $\alpha\gamma\beta$ is derivable from $\alpha A^{(p)}\beta$ only if the grammar contains a production $A^{(q)} = \gamma$ for some $q \geq p$.

Consider the production for meta implication

$$
prop \;=\; prop^{(2)}\ \texttt{==>}\ prop^{(1)} \quad (1) \quad \text{meta implication}
$$

[1]Do not confuse nonterminals *prop* and *logic* with the classes of the same names

[2]The full grammar appears in [34].

It is easy to see how the priorities ensure that meta-implication is right associative.

The productions of the *logic* non-terminal in the above grammar is the *default syntax* for logical types i.e. types in the class `logic`. This is seldom a convenient syntax for logics. Section 5.3 describes Isabelle mechanisms to improve on the concrete object level syntax.

5.2.2 Proof System

The deduction style of the meta logic is Natural Deduction [38]. Deductions use inference rules from a set of assumptions. Each logical operator is equipped with an introduction rule and an elimination rule. Most rules preserve assumptions but the introduction rule for implication allows assumptions to be discarded.

Traditionally the name theorem is reserved for formulas derived from the empty set of assumptions. In Isabelle, however, a theorem is a derived formula together with the set of assumptions from which it was derived. Theorems are written $A \quad \texttt{[}A_1\texttt{,}\ldots\texttt{,}A_n\texttt{]}$ with the assumptions in brackets. Traditionally this would be written as the $A_1, \ldots, A_n \vdash A$ where $\vdash$ is the usual derivation relation.

The rules of the meta logic are listed below. First the rules for implication and logic equivalence:

$$\dfrac{\begin{matrix}[A]\\ \vdots \\ B\end{matrix}}{A\texttt{==>}B}\ \texttt{==>}I \qquad \dfrac{A\texttt{==>}B \quad A}{B}\ \texttt{==>}E$$

$$\dfrac{A\texttt{==>}B \quad B\texttt{==>}A}{A\texttt{==}B}\ \texttt{==}I \qquad \dfrac{A\texttt{==}B \quad A}{B}\ \texttt{==}E$$

Equality is reflective, symmetric and transitive:

$$\dfrac{}{a\texttt{==}a}\ \mathit{refl} \qquad \dfrac{a\texttt{==}b}{b\texttt{==}a}\ \mathit{sym} \qquad \dfrac{a\texttt{==}b \quad b\texttt{==}c}{a\texttt{==}c}\ \mathit{trans}$$

λ expressions can be α- and β-converted and functions are extensional:

$$\dfrac{}{\texttt{(\%}x\,\texttt{.}\,a\texttt{)} \ \texttt{==} \ \texttt{(\%}y\,\texttt{.}\,a[y/x]\texttt{)}}\ \alpha\text{-}\mathit{conv} \quad \text{if } y \text{ is not free in } a$$

$$\dfrac{}{\texttt{((\%}x\,\texttt{.}\,a\texttt{)(}b\texttt{)} \ \texttt{==} \ a[b/x]}\ \beta\text{-}\mathit{conv}$$

$$\frac{f\texttt{(}x\texttt{)}\ \texttt{==}\ g\texttt{(}x\texttt{)}}{f\texttt{==}g}\ ext \quad \text{if } x \text{ is not free in the assumptions, } f \text{ or } g$$

The abstraction and combination rules let conversions be applied to subterms:

$$\frac{a\texttt{==}b}{\texttt{(\%}x\texttt{.}\,a\texttt{)}\ \texttt{==}\ \texttt{(\%}x\texttt{.}\,b\texttt{)}}\ abs \quad \text{if } x \text{ is not free in the assumptions}$$

$$\frac{f\texttt{==}g \quad a\texttt{==}b}{f\texttt{(}a\texttt{)}\ \texttt{==}\ g\texttt{(}b\texttt{)}}\ comb$$

Universal quantification may be introduced and eliminated:

$$\frac{A}{\texttt{!!}x\texttt{.}A}\ \texttt{!!}I \quad \text{if } x \text{ is not free in the assumptions}$$

$$\frac{\texttt{!!}x\texttt{.}A}{A[b/x]}\ \texttt{!!}E$$

Anything is provable by assumption:

$$\frac{}{A\texttt{[}A\texttt{]}}\ assump$$

5.2.3 Unification

The primitive rules of the meta-logic are seldom used to do proofs. Instead proofs use *resolution* as the primary means of deduction. Resolution is useful for both forward proof, combining rules into new rules, and for backwards proof, refining subgoals. Resolution relies on the concept of unification.

Essentially unification is equation solving. Given two expressions X and Y, usually containing free variables, is it possible to find a substitution s instantiating the free variables such that X and Y become equal? Using the notation Xs for the application of substitution we formulate the unification problem as : Xs`=?=`Ys. Solving such an equation in the λ-calculus is complicated by the fact that free variables may need to be instantiated by functions. A solution procedure would have to guess the function `?f` in the unification problem `?`f`(`t`)=?=g(`u_1`, ... ,`u_n`)`. This calls for *higher order unification.*

The problem may have an infinite number of solutions (substitutions s), still the higher order unification procedure by Huet is complete in that it will find all possible solutions. The Isabelle implementation of the procedure generates the solutions as a sequence (lazy list). The Huet procedure relies on type information for its operation (and completeness). The Isabelle procedure solves for type variables as well as term variables. In the presence of the former the Isabelle procedure is incomplete.

For the purpose of unification Isabelle distinguishes between two kinds of free variables: *free variables* which remain fixed in proofs and *unknowns*, which may be instantiated during unification. In other word the unification procedure solves for unknowns only. Unknowns are prefixed with a `?` to tell them from free variables.

5.2.4 Resolution

Resolution is the derived rule:

$$\frac{\texttt{[|}B_1\texttt{;}\ldots\texttt{;}B_n\texttt{|]} \ \texttt{==>}\ B \quad \texttt{[|}A_1\texttt{;}\ldots\texttt{;}A_n\texttt{|]}\ \texttt{==>}\ A}{\texttt{([|}A_1\texttt{;}\ldots\texttt{;}A_{i-1}\texttt{;}B_1\texttt{;}\ldots\texttt{;}B_n\texttt{;}A_{i+1}\texttt{;}\ldots\texttt{;}A_m\texttt{|]}\ \texttt{==>}\ A)s}\ res$$

provided that $A_i s$ `==` Bs.

Resolution can be used in both directions. In forward proofs, resolution combine rules to form new:

$$\frac{\text{rule} \quad \text{rule}}{\text{new rule}}\ res$$

In backwards proofs $\texttt{[|}A_1\texttt{;}\ldots\texttt{;}A_n\texttt{|]}\ \texttt{==>}\ A$ can be seen as a *proof state* with main goal A and subgoals $A_1, \ldots, A_n$. Resolution refines subgoal A_i:

$$\frac{\text{rule} \quad \text{proof state}}{\text{new proof state}}\ res$$

5.3 Object Logics

Encoding an object logic generally consists in declaring types denoting the basic semantic domains of the logic, functions on these domains representing the connectives of the logic and meta-logic axiom representing the rules of the logic.

5.3.1 Semantic Domains and Syntax

Encoding an object logic generally consists in declaring a type to represent formulas of the logic and for each n-ary connective of the logic to declare a n-argument term constant on that type.

Semantically these types denote the meaning of formulas and terms while the functions represent the semantic interpretation of the connectives.

However types also play an important syntactical role. Having types represent the syntactic categories makes it possible to exploit the type-checking of the meta-logic to do syntax checking.

Often a type can serve both purposes being a semantical domain and a representing a syntactic category.

Example The type `o` of the Minimal Logic of Implication is an example of this. It both denotes the set of truth-values and represents the (only) syntactic category.
□

When defining functions to represent connectives the goal is to have an *adequate* representation of the object logic syntax meaning that the representation as λ-terms should be fine grained enough to provide distinct representations of distinct object logic formula and other syntax categories.

We can think of a function $[\![\cdot]\!]$ mapping object logic syntax to it encoded representation as λ-terms of the meta-logic. This function must be adequate in the above sense.

5.3.2 Object Logic Proof Systems

Rules of the object logic are represented as meta logic axioms. Meta-implication is used for logical entailment, meta-quantification expresses generality and meta-equality expresses definition.

The next chapter describes in detail how to encode DC. Below we illustrate the general approach with the recurring example of Minimal Logic of Implication.

Example The inference rule of the Minimal Logic of Implication is Modus Ponens:

$$\dfrac{\begin{matrix} P \\ \vdots \\ Q \end{matrix} \qquad P}{Q}\ MP$$

which is represented as the meta-logical axiom:

```
MP  !!P Q.[| Trueprop(P)==>Trueprop(Q); Trueprop(P)|] ==> Trueprop(Q)
```

This axiom formalizes the justification of the object rule rather than its syntax. We read `Trueprop(P)` as *P is true* and the axiom is read *if firstly Q is true when assuming P is true and secondly P is true, then Q is true.*

Dropping outer-most universal quantification and using implicit coercion by `Trueprop` this becomes more readable:

```
[| P ==> Q; P|] ==> Q
```

□

5.3.3 Faithfulness

Consider a object logic L and let $\mathcal{M}_L$ be the meta-logic obtained by extending $\mathcal{M}$ with types, constants and axioms. The goal of the encoding into $\mathcal{M}_L$ is to have a *faithful* representation of the object logic proof system. This means that a proof in $\mathcal{M}_L$ can be mimicked in L and visa versa in the following sense:

Definition (Faithfulness) Let A be a formula of L and let $[\![\cdot]\!]$ be an adequate function mapping A to a meta-formula $[\![A]\!]$. Then

- $\mathcal{M}_L$ is *sound* if for every $\mathcal{M}_L$-proof of $[\![A]\!]$ there is an L-proof of $[\![A]\!]$.
- $\mathcal{M}_L$ is *complete* if for every L-proof of A there is an $\mathcal{M}_L$-proof of $[\![A]\!]$.
- $\mathcal{M}_L$ is *faithful* if $\mathcal{M}_L$ is both sound and complete.

□

Soundness is the most important of the two conditions while completeness may be sacrificed if the encoding is still powerful enough for practical use.

The standard technique for proving soundness and completeness is by induction on the size of proofs in the logics $\mathcal{M}_L$ and L, respectively. These proofs hinge on the ability to construct meta-deductions formalizing inferences by object logic rules.

Paulson [36] and Frost [16] exemplify the approach on Intuitionistic Proposition Logic and the Operational Logic of Effects, respectively. We base out treatment on their exposition of the basic concepts due to Prawitz [38].

5.3.4 Completeness

Completeness hinges on the representability of the axioms and rules of L.

Definition: Representable rule A rule R of L with premises $P_1, \ldots P_n$ and conclusion C is *representable* iff the meta-theorem $[\![C]\!]$ `[`$[\![P_1]\!]$`,` $\ldots$ `,`$[\![P_n]\!]$`]` can be derived in $\mathcal{M}_L$. We call this meta-theorem the *representation* of R. Axioms are rules with $n = 0$.

□

The proof of completeness reduces to proving the rules and axioms representable. The proof resembles that of deductive power of $\mathcal{IL}_\mathcal{S}$ found in Chapter 2.

Theorem: Completeness The encoding L into $\mathcal{M}_L$ is complete if all rules and axioms of L are representable.

Proof: Assume all axioms and rules representable. The proof is by induction of the size of proofs in L.

If the proof is of size 1 then it must be an axiom of L with is representable by assumption.

For a proof of size $n > 1$ assume the root of the proof is rule R with premises $P_1, \ldots P_n$ and conclusion C. Assume further, as the induction hypothesis, that we have proofs of $[\![P_1]\!]$ to $[\![P_n]\!]$ in $\mathcal{M}_L$. From these proofs we easily construct a proof of $[\![C]\!]$ using the representation of R.

□

5.3.5 Soundness

The proof of soundness essentially is an induction proof over the size of proofs in $\mathcal{M}_L$. The proof reduces, as we will show, to proving soundness of all meta-axioms encoding the rules of L in $\mathcal{M}_L$. The proofs we consider in the induction are a subset of proofs in $\mathcal{M}_L$ called *expanded normal proofs*. These proofs represent a kind of normal form of proofs.

In the following we simplify matters by not considering equality rules and proofs containing such rules. Effectively we identify equal meta-terms and only consider proofs built from introduction and elimination rules for meta-implication and meta-quantification. As a justification of this simplification Frost ([16], page 82) illustrates how the use of `==I` can be replaced with the use of introduction and elimination rules.

First we define the concept of a *branch* of a proof tree. Informally a branch extends from a leaf down the proof tree and ends at the root or at the second premise of the *first* `==>I` node encountered.

Definition: Branch A *branch* of a meta-proof is a path of the proof tree consisting of rule instances $R_1, \ldots, R_n$ such that:

- R_n is the root of the proof tree or the conclusion of R_n is the second premise of an `==>I` rule.

- and for $i < n$, the conclusion of R_i is *not* the second premise of an `==>` rule.

- and R_1 is an axiom

  The *main branch* of a proof is the branch ending in the conclusion of the proof.

□

Using this notion we define a *normalized* proof:

Definition: Normal proof A proof is called *normal* if every branch of the proof is a path of elimination rules followed by a path of introduction rules.
□

Every proof has an equivalent normal proof with the same conclusion [16].

Following the formulas of a branch it is characteristic that they first shrink in size and later grow. Thus branches have a smallest or *minimum* formula:

Definition: Minimum formula A branch has the form $E_1, \ldots E_i, I_{i+1}, \ldots, I_n$ where E_j are elimination rules and I_j are introduction rules. The conclusion of E_i is the *minimal* formula of the branch.
□

Normal proofs may be put in *expanded* normal form [38]. It is characteristic for expanded normal proofs that the minimal formula of every branch is *atomic*.

Definition: Sound rule A meta-axiom A (encoding a rule) is *sound* if for every expanded normal proof Π of $[\![B]\!]$ in which the main branch starts with A, it is possible to construct a proof of B in L, assuming the soundness of all expanded normal proofs smaller than Π.
□

Theorem: Soundness The encoding is sound if all meta-axioms are sound.

Proof: By induction of the size of expanded normal proofs of $[\![B]\!]$, construct a proof of B in L for each such proof. Since $[\![B]\!]$ is atomic it must also be the minimum formula of the main branch. There are two cases:

- Either the branch is empty and $[\![B]\!]$ is an atomic meta-axiom encoding an axiom of L. Then the proof in L is simply the axiom B.
- Or the branch is non-empty and starts with a meta-axiom A but consists only of elimination rules. The existence of a L-proof of B follows from the soundness of the meta-axiom A and the induction hypothesis.

□

Example We prove the `MP` rule of *MLI* sound. With implicit `Trueprop` coercion and expanded chaining the rule looks like:

```
MP  !! P Q. (P ==> Q) ==> (P ==> Q)
```

An expanded normal proof of $[\![B]\!]$ with a main branch starting in `MP` must have the following form:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{!! P Q. (P==>Q) ==> (P==>Q)}}\,\texttt{MP}}{\texttt{!! Q. (}\llbracket A\rrbracket\texttt{==>Q) ==> (}\llbracket A\rrbracket\texttt{==>Q)}}\,\texttt{!!E}}{\texttt{(}\llbracket A\rrbracket\texttt{==>}\llbracket B\rrbracket\texttt{) ==> (}\llbracket A\rrbracket\texttt{==>}\llbracket B\rrbracket\texttt{)}}\,\texttt{!!E} \qquad \dfrac{\begin{matrix}\texttt{[}\llbracket A\rrbracket\texttt{]}\\ \vdots \\ \llbracket B\rrbracket\end{matrix}}{\llbracket A\rrbracket\texttt{==>}\llbracket B\rrbracket}\,\texttt{==>I}}{\llbracket A\rrbracket\texttt{==>}\llbracket B\rrbracket}\,\texttt{==>E} \qquad \begin{matrix}\vdots \\ \llbracket A\rrbracket\end{matrix}}{\llbracket B\rrbracket}\,\texttt{==>E}$$

This proof contains a proof of $\llbracket B\rrbracket$ from $\llbracket A\rrbracket$ and a proof of $\llbracket A\rrbracket$. By assumption these proof have proofs in L. Combine these proof in L with the *MP* rule to get a proof of B in L.
□

5.4 Syntactic Convenience

The syntax of object logics is the λ-calculus. As this is seldom a convenient syntax for logics Isabelle offers a number of mechanisms to improve on this. We describes these mechanisms in the following.

5.4.1 Annotations

Fortunately Isabelle provides a mechanism to control and improve on the concrete syntax of the object logic. Essentially it is possible to extend the grammar with new non-terminals as well as productions for these and existing non-terminals.

The main vehicle for controlling the concrete syntax is *mixfix annotations*. Mixfix annotations are annotations to term constant declarations, and they add a production to the grammar. Consider an annotated constant declaration:

$$con \quad \texttt{::} \quad \texttt{[}t_1\texttt{,} \ldots \texttt{,}t_n\texttt{]} \texttt{ => } t \qquad \texttt{("}w_0 \_ \ldots \_ w_n\texttt{" [}p_1\texttt{,} \ldots \texttt{,}p_n\texttt{] } p\texttt{)}$$

which declares *con* as an n-argument function. w_i is any string and the character _ serve as placeholder for the n arguments to *con*. the production added to the grammar by such an annotation is:

$$A^{(p)} \quad = \quad w_0 A_1^{(p_1)} w_1 A_2^{(p_2)} \ldots A_n^{(p_n)} w_n$$

A and A_i are the non-terminals corresponding to the types t and t_i, respectively. The non-terminal of a logical type (except `prop`) is *logic*. The non-terminal of other types t is t.

The annotation `(infixl` p`)` declares a constant *con* as infix left-associative and is short hand for the mixfix annotation `("`_ *con* _`"` `[`p`,`$p+1$`]` p`)`. Similarly `(infixr` p`)` is short for `("`_ *con* _`"` `[`p`,`p+1`]` p`)` and declares *con* as right-associative. The `binder` annotation is use to obtain a convenient concrete syntax for quantifiers. An example is the annotation for universal quantification in DC. The declaration

```
All           :: ('a => o) => o     (binder "ALL " 10)
```

enables the concrete syntax `ALL x. P(x)` to be used instead of the more cumbersome `All(%x.P(x))`.

Example Using annotations we can improve on the syntax of the Minimal Logic of Implication introduced above. We declare:

```
consts
  Trueprop :: o => prop           ("_" 5)

  "--> "   :: [o, o] => o         (infixr 10)
```

Now we can state the axiom $P \Rightarrow (Q \Rightarrow P)$ simply as `P --> Q --> P`. Notice the annotation for `Trueprop`. It has no concrete syntax, being in effect an implicit coercion from `o` to `prop`. Grammatically one can view this as adding a new production to the *prop* non-terminal:

$$prop \;=\; logic$$

□

5.4.2 Syntax Transformation

Macros and translations are powerful mechanisms complementing the use of annotations. Annotations on their own are only adequate when there is a close relation between the concrete syntax and the internal λ-calculus representation. Macros and translations are applied as steps in the Isabelle parsing and printing process.

The parsing process starts from the string to be parsed. Through application of the lexer and parser (the grammar controlled in part by annotations) a parse tree is produced. The parse tree in then translated into an intermediate Abstract Syntax Tree (AST) form. Macros specify a rewrite system on these ASTs. After AST rewriting, the AST goes through parse translation and type inference to become a term in the internal representation. Parse translations are specified as Standard ML functions and are therefore very general but somewhat cumbersome to program.

The printing process is in the other direction. First print translations are applied to produce an AST. Then macro rewriting takes place and finally the resulting AST is translated to a printable string via print AST translation and token translation.

5.4.3 Macros

Macros specify rewrite rules on ASTs and terms. The syntax for a macro is:

$$\textit{string} \quad \left\{ \begin{array}{c} \texttt{=>} \\ \texttt{<=} \\ \texttt{==} \end{array} \right\} \quad \textit{string}$$

where `=>` specifies a parse rule, `<=` a print rule and `==` both.

A typical use would be to have a not-equal operator in the concrete syntax and expand it to negated equality in the internal λ-calculus representation. The benefit of this would be that deductive procedures working on the internal representation would not have to handle not-equal as a special case. Suppose we have a declaration of equality like

```
"="  :: ['a, 'a] => o    (infixl 50)
```

We would add a *syntax* declaration for the non-equality operator. Syntax declaration resemble constant declarations but only extends the syntax without declaring a new constant.

```
"~=" :: ['a, 'a] => o    (infixl 50)
```

Having extended the syntax we would then use a macro to do the rewriting:

```
"x ~= y" == "not (x = y)"
```

5.4.4 Translations

Parse and print translations are more general than macros but require programming in SML. A translation is a pairing (c, f) of a constant c and an SML function f. A term of the form $c(t_1, \ldots, t_n)$ would trigger a print translation by f replacing the term with $f(t_1, \ldots, t_n)$. Likewise for parse translations on ASTs. Translations are used to provide a convenient syntax for sequences of formulas used in the encoding of DC in Chapter 6.

5.5 Proof Construction

Isabelle supports both forward and backwards proofs in the meta-logic. While forward proof is the basic proof style of Isabelle, backwards proof may be simulated by it and the backwards style is the style most often employed by the Isabelle user. In the following we describe the facilities in Isabelle for forward and backwards proof.

5.5.1 Forward Proof

Isabelles axioms, theorems and object logic rules are represented by the ML type `thm`. The primitive meta-logic rules in Section 5.2.2 are available as ML functions on the `thm` type. Such a function take premises as arguments and return the conclusion of a rule and these functions are the only ways to construct values of type `thm`. The ML type system ensures this. As a consequence the question of soundness of the Isabelle prover is isolated to the soundness of these few functions. Below this approach is exemplified. `cterm` is the type representing *certified* (type-checked) terms.

```
assume        : cterm -> thm
implies_intr  : cterm -> thm -> thm
implies_elim  : thm -> thm -> thm
RSN           : thm * (int * thm ) -> thm
```

`assume` *ct* return the theorem A`[`A`]` where A is the formulas represented by *ct*.

`implies_intr` *ct* A implements the `==>`I rule with *ct* indicating which assumption B of A to discard. The result is the theorem B`==>`A with B removed from the assumptions of A.

`implies_elim` A B implements the `==>`E rule.

`RSN (`B`, (`i`, `A`))` implements the resolution rule *res*, refining the i'th subgoal of A.

Forward proofs are constructed by joining these functions.

5.5.2 Backwards Proof

Backwards proofs are goal-oriented in the sense that they start out from the formula to be proven, the *goal*, and recursively break the goal down into subgoals until eventually all subgoals can be discarded. This process can be simulated using forward proofs.

Say we want to prove an atomic formula A. We would start out from the theorem A`==>`A. This implication represents a *proof state* with the main goal as the conclusion and the subgoals (here only one) as premises. Through resolution we can break the subgoals down and eventually discard them all and be left with A as a theorem thereby having achieved what we set out for.

In case our aim is to prove B`==>`A with B as a premise to A we would still start of from the proof state A`==>`A but allow ourselves to discard subgoals using the theorem B`[`B`]`. This will add B as an assumption to the proof state. Having discarded all subgoals we are left with the proof state A`[`B`]` where we can discard the assumption B and are left with B`==>`A as a theorem.

Isabelle stores both the current and all previous proof steps during a proof in an internal *state list*:

$$[(A_n, \mathcal{B}_n), \ldots, (A_0, [])]$$

A_n is the n'th and current proof state and A_0 is the initial proof state. $\mathcal{B}_i$ are sequences of proof states representing branch points in the proof. n is called the *level* of the proof list.

Backward proofs by resolution employ tactics rather than rules. Tactics are ML functions from theorem to sequences (lazy lists) of theorems. The state list is extended by applying a tactic to the current proof state. The first element in the list returned by the tactic becomes the new current subgoal A_{n+1} and any remaining theorems are stored as $\mathcal{B}_{n+1}$. Backtracking is possible over these alternative proof states.

The main functions for manipulating the proof list are:

```
goal    : theory -> string -> thm list
by      : tactic -> unit
back    : unit -> unit
choplev : int -> unit
qed     : string -> unit
```

The `theory` type is the ML representation of the object logic as a whole: types, syntax, rules. The system support multiple theories (object logics) simultaneously.

`goal` *theory formula* starts a new backwards proof initializing the state list as a side effect. The string *formula* is parsed and type-checked as a formula of the theory *theory*. If *formula* contains premises these are stripped and returned as theorems. The resulting formula A is the goal to be proven and the state list is initialized to $[(A\texttt{==>}A, [])]$.

`by` *tactic* applies *tactic* to the current proof state A_n possibly extending the state list.

`back()` backtracks to the next branch point in the state list making this the new current proof state.

`choplev` i chops off levels of the proof state making i the current level.

`qed` *name* ends a proof started with `goal`. It is checked that the proof state contains no subgoal. Then assumptions are discarded and the resulting theorem is stored in the Isabelle theorem database under the name *name*. The theorem is also bound to the SML variable *name*.

Tactics are essential to backwards proofs. They are used to extend the state list when breaking down subgoals. Examples of tactics are:

```
resolve_tac  : thm list -> int -> tactic
match_tac    : thm list -> int -> tactic
assume_tac   : int -> tactic

res_inst_tac : (string*string) list -> thm -> int -> tactic

no_tac       : tactic
all_tac      : tactic
```

`resolve_tac` *thms* i tries to resolve the i'th subgoal with theorems in the list *thms*.

`match_tac` *thms* i tries to match the i'th subgoal with theorems in the list *thms*. Matching is like resolution but *never* instantiates unknowns in the proof state.

`assume_tac` i tries to solve the i'th subgoal of the current proof state by assumption.

`res_inst_tac` *insts thm* i first instantiates unknowns in *thm* with *insts* and then tries to resolve the i'th subgoal with the resulting theorem. A pair (v, e) is *insts* is an instantiation of unknown v with the expressions v. v are typed checked in the context of the i subgoal.

`no_tac` returns the empty sequence of proof states.

`all_tac` *thm* returns the singleton sequence consisting of *thm*.

Tacticals are used to program proof search. They are functions on tactics. For example the tacticals `THEN` and `ORELSE` provide sequencing and alternation:

```
THEN   : tactic * tactic -> tactic
ORELSE : tactic * tactic -> tactic
```

tac_1`THEN` tac_2 is the sequential composition of two tactics. Applied to a proof state, it returns all states reachable in two steps by applying tac_1 followed by tac_2. `no_tac` is the zero element of `THEN` while `all_tac` is the identity element.

tac_1`ORELSE` tac_2 makes a choice between two tactics. Applied to a state, it tries tac_1 and returns the result if non-empty; if tac_1 fails then it uses tac_2. `no_tac` is the identity element of `ORELSE`.

5.6 Simplification

Isabelle provides a simplification package which can be setup for a variety of logics. Current setups are for FOL, HOL etc. The package (an SML structure) exports tactics like:

```
type simpset

simp_tac : simpset -> int -> tactic
Simp_tac : int -> tactic
```

`simpset` contains the setup for the logic including the rewriting rules used for simplification.

`simp_tac`*ssi* sets up the simplifier according to *ss* and applies simplification to the *i*'th subgoal.

`Simp_tac` There is a notion of a *current simpset css* associated with a theory. `Simp_tac` behaves like `simp_tac`*css*.

5.6.1 Simplification process

Simplification as supported by the simplifier is a process of meta-level rewriting and theorem proving: First the subgoal is rewritten using rewrite rules. Any new subgoals stemming from the application of conditional rewrite rules is then attacked by the *subgoaler*, a configurable tactic. Often the subgoaler simply applies simplification on the new subgoals. After rewriting the *solver*, again a configurable tactic, is applied to the subgoal. The solver often is a simple tactic proving goals like `True` or a=b. If the solver fails the *looper* tactic is applied. The looper is often used to do a case split and then to re-apply simplification.

The rewrite rules, subgoaler, solver and looper are all part of a setup and stored in a simpset.

```
empty_ss : simpset

setsubgoaler : simpset * (simpset -> int -> tactic) -> simpset
setSolver : simpset * (thm list -> int -> tactic) -> simpset
setlooper : simpset * (int -> tactic) -> simpset
```

5.6.2 Rewrite Rules

At the object level rewrite rules involve object level equivalence or equality like the rules:

```
|- A & A <-> A
|- x + zero = x
```

Rules like these are added to a simpset by:

```
addsimps : simpset * thm list -> simpset
```

Internally the simplifier employs meta-level rewriting rules. Meta-level rewrite rules are theorems of equality of the form A==B or `[|` C_1 ; ... ; C_n `|] ==>`A==B both theorems rewriting A into B.

Object rules like the above thus need to be converted into meta-level rules to be usable in rewriting. This requires a connection between the object level operators and meta-equality established as basic rules:

```
iff_reflection   |- A<->B ==> (A==B)
eq_reflection    |- x=y ==> (x==y)
```

Resolution with the reflective rules easily produce the desired meta-level rewrite rules:

```
A & A == A
x + zero == x
```

This preprocessing of object level rules into meta-level rewrite rules is part of the simplifier setup and supplied as the second argument to `setmksimps`:

```
setmksimps : simpset * (thm -> thm list) -> simpset
```

5.6.3 Simplification Procedures

Adding rewrite rules using `addsimps` allows for rewriting with a fixed set of rules. The simplifier allow greater flexibility using simplification procedures with are functions generating context dependent rewrite rules on the fly. The simplifier may call a simplification procedure on the current term redex to get specific (proven) rewrite rules. Simplification procedures are added to a simpset using:

```
type simproc

addsimprocs : simpset * simproc list -> simpset
```

5.6.4 Permutative Rewrite Rules

In *permutative* rewrite rules the left and right sides are equal up to renaming of variables. Commutative rules are thus permutative.

Permutative rules may cause non-termination when rewriting. To avoid this the simplifier uses *ordered* rewriting. Given a total ordering on terms the simplifier will only apply a permutative rule if the right hand side is less than the left hand side.

A lexicographical ordering is used as default but it may be changed:

```
settermless : simpset * (term * term -> bool) -> simpset
```

5.7 Summary

We have presented the type system and meta-logic of Isabelle together with the facilities Isabelle provides for encoding object logics. We make use of these feature in the following chapter where we encode Duration Calculus.

Chapter 6

Isabelle/DC

This chapter will present the encoding of DC into Isabelle. The starting point is the sequent style logics $\mathcal{IL}_\mathcal{S}$ and $\mathcal{DC}_\mathcal{S}$ presented in Chapter 2. The aim is to encode the syntax and proof rules of these logics in an faithful way as defined in section 5.3.3.

The encoding extends the basic Isabelle theory `Pure` with a hierarchy of Isabelle theories. A theory is simply a named collection of declared types, values and axioms. First we encode sequences of formulas necessary to represent sequents as the theory `Sequence`. The we encode Predicate Logic of section 2.1.4 as the theory `PL` and extend this with a theory of real arithmetic `RA`. We do these encodings with an eye towards the later extension into `IL`, the Interval Logic of section 2.1.4. On top of `IL` we build `DC`, the Duration Calculus of section 2.2.4, by extension. We also encode the Extended Duration Calculus of section 2.4 as the theory `EDC` and the standard forms of section 2.5 as the theory `FB`. The complete hierarchy is depicted in figure 6.1. A line in the figure means that the upper theory extends the connected lower theory.

In the following we devote a section to each theory and describe the syntax and proof rules that the theory encodes. Recall that the aim is to present an adequate syntax and a faithful proof system. Appendix A contains the theories in full.

6.1 Sequence

The theory `Sequence` contains the encoding of the sequences of formulas needed to build sequents[1]. The problem is to achieve a convenient syntax while maintaining the associativity of the comma separator during unification. This is done by having separate external and internal syntax linked via syntax transformation as introduced in section 5.4.2.

[1] This is copied to a large extent from the Isabelle distribution.

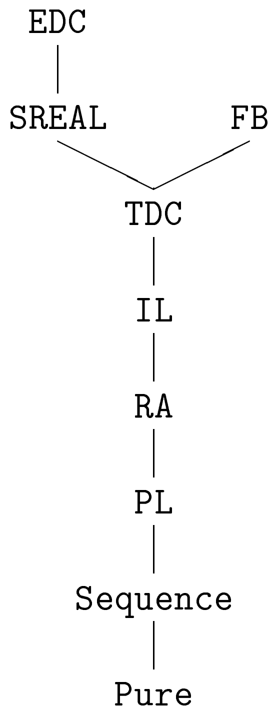


Figure 6.1: Hierarchy of Isabelle/DC theories

Initially we declare the type of object logic formulas `o` and the classes to which this type belongs. This is needed as the elements of a sequence are formulas.

```
classes
  term < logic
  fterm < term

types
  o

arities
  o :: fterm
```

Later in this chapter as we move up in the hierarchy of theories we will eventually introduce three layers of term syntax. The reason for having these layers will be clear when we encode DC in Section 6.5. The class `fterm` will contain what we call *flexible* terms and formulas (`o`) are flexible terms. Semantically, the meaning of flexible terms may depend on the context interval. The class `term` is the superclass of terms and will contain all term types. We will reason about types in `term` so we make `term` a subclass of the predefined class `logic`.

Internally a sequence $A_1, \ldots, A_n$ is build using the constant `Seq'` as a cons-like constructor:

```
types
 seq'

consts
 Seq'           :: "[o,seq']=>seq'"
```

and the representation of $A_1, \ldots, A_n$ is `%x.Seq'(`A_1`,Seq'(`A_2`, ... ,Seq'(`A_n`,x)...))`. The type of an internal sequence is thus `seq' => seq'` and a sequence can be seen as a function concatenating a sequence at the end of a list of elements. The functional representation eliminates the need for a constructor of the empty sequence which is simply represented by `%x.x`.

The advantage of the functional representation is that unifying a sequence with the term `%x.?f(?g(x))` gives all the ways of splitting it into two sub-sequences.

Externally sequences use a more conventional representation:

```
types
  seq seqobj seqcont

syntax
 SeqEmp         :: "seq"                                ("")
 SeqApp         :: "[seqobj,seqcont] => seq"            ("__")

 SeqContEmp     :: "seqcont"                            ("")
 SeqContApp     :: "[seqobj,seqcont] => seqcont"        (",/ __")

 SeqO           :: "o => seqobj"                        ("_")
 SeqId          :: "id => seqobj"                       ("$_")
 SeqVar         :: "var => seqobj"                      ("$_")
```

The type of sequences is `seq` and the type of sequence elements is `seqobj`. Sequence elements can be either an object logic formula of type `o` or variables (or unknowns) ranging over sequences. The latter must be distinguished somehow from variables ranging over formulas which is why they are prefixed with a `$`

A sequence containing the formula `A` is written `$Gamma,A,$Delta` where `Gamma` and `Delta` are sequence variables. Using syntax transformations this is converted to the internal representation `%x.Gamma(Seq'(A, Delta(x)))`.

We state without proof that this representation is faithful to the sequences of $\mathcal{IL}_\mathcal{S}$ defined in Chapter 2.

6.2 PL

The PL theory encodes the sequent style proof system for predicate logic found in chapter 2. The theory is a modification of the one by Paulson which appears in the Isabelle distribution. Modification is necessary to correctly encode the rules for quantification in the presence of flexible and chopped formulas.

6.2.1 Syntax

The theory first declares a new class `rterm`, the class of *rigid* terms and the type of rigid formulas `ro`. Rigid terms constitute our second layer of syntax, flexible terms was the first. Semantically, the meaning of rigid terms are independent of the any context interval.

```
classes
  rterm < term

types
  ro

arities
  ro :: rterm
```

We declare the class `term` as default, meaning that type variables range over types of this class if no sort is explicitly specified.

```
default
  term
```

The `Sequents` theory declared the type of DC formulas `o` and sequences of this type `seq`. PL declares the injection of sequents (pairs of sequences) into the meta logical type `prop`. There are two of these injections, an external and an internal. The conversion between them is done by syntax transformations. The external injection separates the left and the right sequences with the `|-` sign resembling the turnstile symbol $\vdash$.

```
  "@Trueprop"    :: "[seq, seq] => prop"  ("_ |- _")
  "Trueprop"     :: "[seq', seq'] => prop"
```

The syntax of propositions is encoded below. We have two layers of logical formulas (`o` and `ro`) and would like to use the same concrete syntax for both layers. Isabelle does not allow overloading, instead polymorphism is used. First the class `prp` is declared and both formulas types inserted into it. We now declare the constants encoding the propositional syntax as polymorphic over types in `prp`. This effectively gives us the same syntax in both layers. We will use this trick again but unfortunately its usefulness is limited as we shall see below when declaring equality.

```
classes
  prp

arities
  o :: prp
  ro :: prp

consts
  True,False    :: 'a::prp
  Not           :: 'a::prp => 'a              ("~ _" [40] 40)
  "&"           :: ['a::prp,'a] => 'a         (infixr 35)
  "|"           :: ['a::prp,'a] => 'a         (infixr 30)
  "-->","<->"   :: ['a::prp,'a] => 'a         (infixr 25)
```

We use higher order syntax for the quantifiers. In a quantification $\forall x \cdot P(x)$ we can abstract $P(x)$ into the function $\lambda x \cdot P(x)$ and view the quantification $\forall$ as the function that checks if $\lambda x \cdot P(x)$ is true for all x. The `binder` directive allow us to used the syntax $\forall x \cdot P(x)$ instead of the cumbersome $\forall(\lambda x \cdot P(x))$ Notice that quantification is possible over both rigid and flexible terms.

```
  All           :: ('a => o) => o             (binder "ALL " 10)
  Ex            :: ('a => o) => o             (binder "EX " 10)
```

Equality need two operators, one at each syntax level. The trick using polymorphism cannot be used as the argument and result types are not the same. We institute the convention of prefixing operators at the rigid level with a colon (`:`).

```
  "="           :: ['a::fterm,'a] => o        (infixl 50)
  ":="          :: ['a::rterm,'a] => ro       (infixl 50)
```

Injection of rigid terms into flexible terms is possible through lifting functions `n`t, where t is some type. Here we lift formulas with the function `no`:

```
  no            :: ro => o
```

6.2.2 Proof Rules

The rules of PL are grouped into the structural, proposition and quantification rules. Recall that preceding a variable with a `$` makes it range over sequences.

Structural Rules The structural rules appear below. They allow contraction, thinning and exchange of sequences. Structural rules for single formulas are derived.

$$\frac{\Gamma, \Gamma'' \vdash \Delta}{\Gamma, \Gamma', \Gamma'' \vdash \Delta}\ \mathrm{T}\vdash \qquad \frac{\Gamma \vdash \Delta, \Delta''}{\Gamma \vdash \Delta, \Delta', \Delta''}\ \vdash\mathrm{T}$$

```
thinLS  $H, $G |- $E ==> $H, $S, $G |- $E
thinRS  $H |- $E, $F ==> $H |- $E, $S, $F
```

$$\frac{\Gamma, \Gamma', \Gamma', \Gamma'' \vdash \Delta}{\Gamma, \Gamma', \Gamma'' \vdash \Delta}\ \mathrm{C}\vdash \qquad \frac{\Gamma \vdash \Delta, \Delta', \Delta', \Delta''}{\Gamma \vdash \Delta, \Delta', \Delta''}\ \vdash\mathrm{C}$$

```
contLS  $H, $S, $S, $G |- $E ==> $H, $S, $G |- $E
contRS  $H |- $E, $S, $S, $F ==> $H |- $E, $S, $F
```

$$\frac{\Gamma, \Gamma', \Gamma'', \Gamma''' \vdash \Delta}{\Gamma, \Gamma'', \Gamma', \Gamma'' \vdash \Delta}\ \mathrm{E}\vdash \qquad \frac{\Gamma \vdash \Delta, \Delta', \Delta'', \Delta'''}{\Gamma \vdash \Delta, \Delta'', \Delta'', \Delta'''}\ \vdash\mathrm{E}$$

```
exchLS  $H, $R, $S, $G |- $E ==> $H, $S, $R, $G |- $E
exchRS  $H |- $E, $R, $S, $F ==> $H |- $E, $S, $R, $F
```

Propositional rules Recall the propositional rule Ax:

$$\frac{}{A \vdash B}\ \mathrm{Ax} \qquad \text{if } A \text{ and } B \text{ are syntactically equal up to renaming of bound variables.}$$

```
$H, A, $G |- $E, A, $F  basic
```

The side condition of the Ax rule express that A and B must be α-equal. This is implicit in the `basic` rule because unification will identify α-equal terms.

The other propositional rules are straightforward to encode. Multiple premises are encoded using chained meta-implication.

$$\frac{\Gamma, A, B, \Gamma' \vdash \Delta}{\Gamma, A \wedge B, \Gamma' \vdash \Delta}\ \wedge\vdash \qquad \frac{\Gamma \vdash \Delta, A, \Delta' \qquad \Gamma \vdash \Delta, B, \Delta'}{\Gamma \vdash \Delta, A \wedge B, \Delta'}\ \vdash\wedge$$

```
conjL  $H, P, Q, $G |- $E ==> $H, P & Q, $G |- $E
conjR  [| $H|- $E, P, $F;  $H|- $E, Q, $F |] ==> $H|- $E, P&Q, $F
```

$$\frac{\Gamma, A, \Gamma' \vdash \Delta \qquad \Gamma, B, \Gamma' \vdash \Delta}{\Gamma, A \vee B, \Gamma' \vdash \Delta} \vee\vdash \qquad \frac{\Gamma \vdash \Delta, A, B, \Delta'}{\Gamma \vdash \Delta, A \vee B, \Delta'} \vdash\vee$$

```
disjL  [| $H, P, $G |- $E;  $H, Q, $G |- $E |] ==> $H, P|Q, $G |- $E
disjR  $H |- $E, P, Q, $F ==> $H |- $E, P|Q, $F
```

$$\frac{\Gamma, \Gamma' \vdash \Delta, A \qquad \Gamma, B, \Gamma' \vdash \Delta}{\Gamma, A \Rightarrow B, \Gamma' \vdash \Delta} \Rightarrow\vdash \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \Rightarrow B, \Delta} \vdash\Rightarrow$$

```
impL  [| $H,$G |- $E,P;  $H, Q, $G |- $E |] ==> $H, P-->Q, $G |- $E
impR  $H, P |- $E, Q, $F ==> $H |- $E, P-->Q, $F
```

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

```
notL  $H, $G |- $E, P ==> $H, ~P, $G |- $E
notR  $H, P |- $E, $F ==> $H |- $E, ~P, $F
```

$$\frac{}{\Gamma, \mathit{false}, \Gamma' \vdash \Delta} \mathit{false}\vdash$$

```
FalseL  $H, False, $G |- $E
```

$$\frac{\Gamma \vdash \Delta, A \qquad \Gamma, A \vdash \Delta}{\Gamma \vdash \Delta} \mathrm{Cut}$$

```
cut  [| $H |- $E, P;  $H, P |- $E |] ==> $H |- $E
```

The constant *true* and logical equivalence are encoded as definitions:

```
True_def "True == False-->False"
iff_def  "P<->Q == (P-->Q) & (Q-->P)"
```

Quantifier rules The quantifier rules and their meta-logic axiom will not resemble each other like the structural and propositional rules did. The encoding of the side conditions will be different.

The quantifier rules have syntactic provisos requiring formulas and terms to be chop free or rigid. To formalize these concepts we declare meta-logic predicates `CF` and `RT` recognizing such formulas and terms. A number of inference rules are defined for these predicates, we present these later.

```
CF            :: o => prop          ("CF _")
RT            :: 'a => prop         ("RT _")
```

Side conditions requiring some term `t` to be rigid are encoded in the rule by adding a premise `RT t` and likewise for chop free formulas.

The side condition that "t is any term free for x in A" is implicit in the encoding. Substitution $A(t/x)$ is encoded as application `A(t)` and later β-reduction will involve renaming of variables to avoid capture.

$$\frac{\Gamma, A(t/x), \Gamma', (\forall x \cdot A) \vdash \Delta}{\Gamma, (\forall x \cdot A), \Gamma' \vdash \Delta} \forall\vdash \quad \begin{array}{l} t \text{ is any term free for } x \text{ in } A \text{ and} \\ \text{either } t \text{ is rigid} \\ \text{or } A \text{ is chop free and } x \text{ does not occur} \\ \text{in any duration term of } A \end{array}$$

```
allLRT  [| $H, P(t), $G, ALL x. P(x) |- $E; RT t |] ==>
           $H, ALL x. P(x), $G |- $E
allLCF  [| $H, P(t), $G, ALL x. P(x) |- $E; CF(P(t)) |] ==>
           $H, ALL x. P(x), $G |- $E
```

There are two axioms for the $\forall\vdash$ rule: one for rigid t and one for chop free A. Each rule has a corresponding premise added.

$$\frac{\Gamma \vdash \Delta, A, \Delta'}{\Gamma \vdash \Delta, (\forall x \cdot A), \Delta'} \vdash\forall \quad x \text{ does not occur free in the conclusion}$$

```
allR  (!!x. RT x ==> $H |- $E, P(x), $F) ==>
             $H |- $E, ALL x. P(x), $F
```

The eigenvariable condition of the $\vdash \forall$ rule is not formalized literally. In the axiom `P` does not denote the quantified formula A but its meaning, a truth-valued function. The axiom states that if `$H |- $E, P(t), $F` is true for all rigid terms `t` then `$H |- $E, ALL x. P(x), $F` is true.

The encodings of the rules for existential quantification resemble the encodings of the rules for universal quantification. We state these without comment:

$$\frac{\Gamma \vdash \Delta, A(t/x), \Delta', (\exists x \cdot A)}{\Gamma \vdash \Delta, (\exists x \cdot A), \Delta'} \vdash \exists$$

t is any term free for x in A and either t is rigid or A is chop free and x does not occur in any duration term of A

```
exRRT  [| $H |- $E, P(x), $F, EX x. P(x); RT x |] ==>
          $H |- $E, EX x. P(x), $F
exRCF  [| $H |- $E, P(x), $F, EX x. P(x); CF(P(x)) |] ==>
          $H |- $E, EX x. P(x), $F
```

$$\frac{\Gamma, A, \Gamma' \vdash \Delta}{\Gamma, (\exists x \cdot A), \Gamma' \vdash \Delta} \exists \vdash$$

x does not occur free in the conclusion

```
exL  (!!x. RT x ==> $H, P(x), $G |- $E) ==> $H, EX x. P(x), $G |- $E
```

Side conditions To reason about syntactic properties of formulas and terms we declare meta-logic axioms like:

```
CFno      CF(no(x))
CFtrue    CF(True)
CFfalse   CF(False)
CFnot     CF(A) ==> CF(~ A)
CFand     [| CF(A); CF(B) |] ==> CF(A&B)
CFor      [| CF(A); CF(B) |] ==> CF(A|B)
CFimp     [| CF(A); CF(B) |] ==> CF(A-->B)
CFiff     [| CF(A); CF(B) |] ==> CF(A<->B)

CFeq      CF(x = y)

CFall     CF(A(x)) ==> CF(ALL x. A(x))
CFex      CF(A(x)) ==> CF(EX x. A(x))
```

and

```
RTrterm   RT(x::'a::rterm)

RTno      RT(no(x))

RTtrue    RT(True::o)
RTfalse   RT(False::o)
```

```
RTnot    RT(A::o) ==> RT(~ A)
RTand    [| RT(A::o); RT(B) |] ==> RT(A&B)
RTor     [| RT(A::o); RT(B) |] ==> RT(A|B)
RTimp    [| RT(A::o); RT(B) |] ==> RT(A-->B)
RTiff    [| RT(A::o); RT(B) |] ==> RT(A<->B)

RTall    (RT x ==> RT(A(x))) ==> RT(ALL x. A(x))
RTex     (RT x ==> RT(A(x))) ==> RT(EX x. A(x))

RTeq     [| RT x; RT y |] ==> RT(x=y)
```

The form of these rules are convenient for resolution. Notice that `RTall` expresses that if we can prove `A(x)` is rigid by assuming that `x` is rigid then so is `ALL x.A(x)`. We add rules like these for every new connective which preserves rigidity and chop-freeness.

Lifing The lifting function `no` is axiomatized:

```
notrue   True == no(True)
nofalse  False == no(False)
nonot    ~no(A) == no(~ A)
noand    no(A) & no(B) == no(A & B)
noor     no(A) | no(B) == no(A | B)
noimp    no(A) --> no(B) == no(A --> B)
noiff    no(A) <-> no(B) == no(A <-> B)

noeq_o   no(A) = no(B) == no(A := B)
```

Equality The rule of reflexivity of equality is easily encoded. Although the `refl` looks more restrictive it will actually resolve for any a and b which are α-equal.

$$\frac{}{\Gamma \vdash a = b, \Delta}\ \texttt{refl} \qquad \begin{array}{l}a \text{ and } b \text{ are syntactically equal}\\ \text{up to renaming of bound variables}\end{array}$$

```
refl  $H |- $E, a=a, $F
```

The rules for replacement requires further considerations. Recall the rules:

$$\frac{}{a_1 = b_1, \ldots, a_n = b_n \vdash f(a_1, \ldots, a_n) = f(b_1, \ldots, b_n)}\ \texttt{fun\_repl}$$

$$\frac{}{a_1 = b_1, \ldots, a_n = b_n, G(a_1, \ldots, a_n) \vdash G(b_1, \ldots, b_n)}\ \texttt{pred\_repl}$$

These rules are formulated using the concepts of function and predicate symbols (f and G, respectively) for which we have no representation in the encoding. Instead we encode the rule:

```
repl  CF A(x) ==> $I, a=b, $H, A(a), $G |- $E, A(b), $F
```

where `A` may be any formula. The rule `pred_repl` can be seen as the special case of `repl` where `A` is the predicate G.

To avoid unsoundness of the rule when extending `PL` to create Interval Logic the formula `A` must be chop-free. To see why consider the formula $A(x) \hat{=} (\neg(\ell = x)) \frown true$ and the sequent:

$$\frac{}{2 = \ell, (\neg(\ell = 2)) \frown true \vdash (\neg(\ell = \ell)) \frown true}$$

$A(x)$ is not chop-free. If we remove the premise the sequent would become a theorem but it is not valid because the consequent is always false while the antecedent is true for $\ell = 2$.

Restricting `A` in the `repl` rule to only having one parameter is not a limitation. We can derive rules for any n and illustrate the approach for $n = 2$ by the following derivation where thinning steps are suppressed.

$$\dfrac{\dfrac{\dfrac{}{a_1 = b_1, (\lambda x \cdot G(x)(b_2))(a_1) \vdash (\lambda x \cdot G(x)(b_2))(b_1)}\,\texttt{repl}}{a_1 = b_1, G(a_1)(b_2) \vdash G(b_1)(b_2)} \qquad \dfrac{}{a_2 = b_2, G(a_1)(a_2) \vdash G(a_1)(b_2)}\,\texttt{repl}}{a_1 = b_1, a_2 = b_2, G(a_1)(a_2) \vdash G(b_1)(b_2)}\,\texttt{Cut}$$

Because `repl` is more general than `pred_repl` we do not need to encode `fun_repl` as an axiom. Instead we can easily derive the rule:

```
eq_repl  $H, a=b, $G |- $E, f(a) = f(b), $F
```

Recall the substitution rule:

$$\frac{}{\Box(a = b), A(a) \vdash A(b)}\,\texttt{subst}$$

We mentioned in Chapter 2 that this rule could be derived for $\mathcal{IL}_\mathcal{S}$ by a meta-theoretical argument rather than in the logic itself. This means that if we want the convenience of the rule in the encoding we will have to add axioms allowing the rule to be derived.

The axioms we add state that equality and equivalence are related to meta-equality by reflection:

```
eq_reflection   |- x=y ==> (x==y)
iff_reflection  |- A<->B ==> (A==B)
```

The axioms are very convenient when we consider rewriting and simplification in a later section. They also allow us to derive the rule:

```
subst  |- a=b ==> $H, A(a), $G |- $E, A(b), $F
```

in `PL` and the rule:

```
Box_subst  |- $I, [](a=b), $H, A(a), $G |- $E, A(b), $F
```

in `IL` the encoding of Interval Logic presented in a later.

6.3 RA

On top of `PL` we build `RA`, the theory of real arithmetic. Real arithmetic is needed for the duration terms of DC. The primary purpose of this theory is to provide the syntax for arithmetic terms. The reasoning support we plan for this theory is to be done by a decision procedure described in Chapter 8. We do however encode the axioms of section 2.1.4 in the theory.

6.3.1 Syntax

Flexible real terms have the type `real` and rigid terms type `rreal`. `arith` is the class over which we declare the real arithmetic operators. This makes them available in both term levels.

```
classes
  arith

default
  arith

types
  real
  rreal

arities
  real :: arith
  rreal :: arith
  real :: fterm
  rreal :: rterm
```

The syntax of real terms is:

```
zero          :: 'a                          ("0")
one           :: 'a                          ("1")

"+"           :: ['a, 'a] => 'a              (infixl 60)
"-"           :: ['a, 'a] => 'a              (infixl 60)
"--"          :: 'a => 'a                    ("-- _" [80] 80)
"*"           :: ['a, 'a] => 'a              (infixl 70)
"op /"        :: ['a, 'a] => 'a              ("(_ '// _)" [70,71] 70)

"<"           :: [real, real] => o           (infixr 50)
"<="          :: [real, real] => o           (infixr 50)
">"           :: [real, real] => o           (infixr 50)
">="          :: [real, real] => o           (infixr 50)

":<"          :: [rreal, rreal] => ro        (infixr 50)
":<="         :: [rreal, rreal] => ro        (infixr 50)
":>"          :: [rreal, rreal] => ro        (infixr 50)
":>="         :: [rreal, rreal] => ro        (infixr 50)

nr            :: rreal => real
```

`nr` is the lifting function on reals. There is no syntax for real numerals at present. Although we are confident that this can be done it will have to be investigated further. However, we do provide syntax for the digits which is all we will need:

```
syntax
  two           :: 'a                          ("2")
  three         :: 'a                          ("3")
  four          :: 'a                          ("4")
  five          :: 'a                          ("5")
  six           :: 'a                          ("6")
  seven         :: 'a                          ("7")
  eight         :: 'a                          ("8")
  nine          :: 'a                          ("9")

translations
  "2" == "1+1"
  "3" == "2+1"
  "4" == "3+1"
  "5" == "4+1"
  "6" == "5+1"
  "7" == "6+1"
```

```
"8" == "7+1"
"9" == "8+1"
```

6.3.2 Proof Rules

The rules of RA are simply the field and order axioms of real arithmetic stated as sequents. The axioms are:

Literals

```
zero_ne_one          |- ~ (0 = 1)
zero_lt_one          |- 0 < 1
```

Field Axioms

```
commutative_add      |- x + y = y + x
associative_add      |- x + (y + z) = (x + y) + z
identity_add         |- x + 0 = x
inverse_add          |- x + --x = 0
minus_add            |- x - y = x + --y
commutative_mult     |- x * y = y * x
associative_mult     |- x * (y * z) =  (x * y) * z
identity_mult        |- 1 * x = x
inverse_mult         |- ~(x=0) --> x * (1 / x) = 1
div_def              |- ~(x=0) --> y / x = y * (1 / x)
distributive         |- x * (y + z) = (x * y) + (x * z)
```

Order axioms

```
posreal_add_closed   |- x > 0 & y > 0 --> x + y > 0
posreal_mult_closed  |- x > 0 & y > 0 --> x * y > 0
posreal_neg          |- x > 0 --> ~(--x > 0)
trichotomy           |- x > 0 | x = 0 | 0 > x
```

Rigid terms We also encode what it means to be a rigid real term and axiomatize `nr`. The following is an except:

```
RTnr          RT(nr(x))

RTzero        RT(0)
RTone         RT(1)
RTadd         [| RT(x); RT(y) |] ==> RT(x+y)
RTlt          [| RT(x); RT(y) |] ==> RT(x<y)
```

```
nrzero       0 == nr(0)
nrone        1 == nr(1)
nradd        nr(x) + nr(y) == nr(x + y)
nolt         nr(x) < nr(y) == no(x :< y)
```

6.4 IL

The `IL` theory defines Interval Logic as an extension of theory `RA`.

6.4.1 Syntax

We extend the syntax of formulas with the modalities and the interval length term `len`.

```
"^"           :: [o, o] => o                    (infixr 55)
dia           ::  o => o                        ("<>_" [50] 50)
box           ::  o => o                        ("[]_" [50] 50)
len           :: real
```

Diamond and box modalities are abbreviations defined using meta-level equality:

```
dia_def   "<>A == True^A^True"
box_def   "[]A == ~(<>(~A))"
```

6.4.2 Proof Rules

The IL axioms are:

$$\frac{}{\vdash \ell \geq 0}\ \texttt{A0}$$

```
A0  |- len >= 0
```

$$\frac{}{\vdash A \frown B_1 \wedge \neg(A \frown B_2) \Rightarrow A \frown (B_1 \wedge \neg B_2)}\ \texttt{A1a}$$

$$\frac{}{\vdash A_1 \frown B \wedge \neg(A_2 \frown B) \Rightarrow (A_1 \wedge \neg A_2) \frown B}\ \texttt{A1b}$$

```
A1a  |- A^B & ~(A^C) --> A^(B & ~C)
A1b  |- A^B & ~(C^B) --> (A & ~C)^B
```

$$\frac{}{\vdash A \frown (B \frown C) \Leftrightarrow (A \frown B) \frown C} \; \texttt{A2}$$

```
A2  |- (A^B)^C <-> A^B^C
```

$$\frac{}{\vdash A \frown B \Rightarrow A} \; \texttt{Ra} \quad A \text{ is rigid}$$

$$\frac{}{\vdash A \frown B \Rightarrow B} \; \texttt{Rb} \quad B \text{ is rigid}$$

```
Ra  RT(A) ==> |- A^B --> A
Rb  RT(B) ==> |- A^B --> B
```

$$\frac{}{\vdash (\exists x \cdot A) \frown B \Rightarrow (\exists x \cdot A \frown B)} \; \texttt{Ba} \quad x \text{ does not occur free in } B$$

$$\frac{}{\vdash A \frown (\exists x \cdot B) \Rightarrow (\exists x \cdot A \frown B)} \; \texttt{Bb} \quad x \text{ does not occur free in } B$$

```
Ba  |- (EX x. A(x))^B --> (EX x. A(x)^B)
Bb  |- A^(EX x. B(x)) --> (EX x. A^B(x))
```

$$\frac{}{\vdash (\ell = x) \frown A \Rightarrow \neg((\ell = x) \frown \neg A)} \; \texttt{L1a}$$

$$\frac{}{\vdash A \frown (\ell = x) \Rightarrow \neg(\neg A \frown (\ell = x))} \; \texttt{L1b}$$

```
L1a  RT x ==> |- (len=x)^A --> ~((len=x)^(~A))
L1b  RT x ==> |- A^(len=x) --> ~((~A)^(len=x))
```

Notice the axiom `L2` where `x` range over rigid terms only, mimicking that x is a free global variable in the formulation of the same axiom in $\mathcal{IL}_{\mathcal{S}}$. This is the reason for adding the precondition `RT x`. Such preconditions need to be added to axioms and rules containing free term variables.

$$\frac{}{\vdash x \geq 0 \wedge y \geq 0 \Rightarrow (\ell = x + y \Leftrightarrow (\ell = x) \frown (\ell = y))} \; \texttt{L2}$$

```
L2  [| RT x; RT y |] ==>
      |- x >= 0 & y >= 0 --> ((len=x+y) <-> (len=x)^(len=y))
```

$$\frac{}{\vdash (\ell = 0) \frown A \Leftrightarrow A}\ \texttt{L3}$$

$$\frac{}{\vdash A \frown (\ell = 0) \Leftrightarrow A}\ \texttt{L4}$$

```
L3  |- A --> A^(len=0)
L4  |- A --> (len=0)^A
```

The IL rules are:

$$\frac{\Gamma \vdash \Delta, \Box(A \Rightarrow B), \Delta'}{\Gamma \vdash \Delta, A \frown C \Rightarrow B \frown C, \Delta'}\ \texttt{monotone\_chop}$$

$$\frac{\Gamma \vdash \Delta, \Box(A \Rightarrow B), \Delta'}{\Gamma \vdash \Delta, C \frown A \Rightarrow C \frown B, \Delta'}\ \texttt{chop\_monotone}$$

```
monotone_chop  $G |- $E, [](A-->B), $F  ==>  $G |- $E, A^C-->B^C, $F
chop_monotone  $G |- $E, [](A-->B), $F  ==>  $G |- $E, C^A-->C^B, $F
```

$$\frac{\Gamma^{\Box} \vdash A, \Delta^{\Diamond}}{\Gamma \vdash \Box A, \Delta}\ \texttt{boxR}$$

```
boxR  [| $G |L> $G'; $E |R> $E'; $F |R> $F';
         $G' |- $E', P, $F'|]  ==>  $G |- $E, []P, $F
```

`$G |L> $G'` and `$F |R> $F'` are new syntactical relations on sequences encoding the effect of $\Gamma^{\Box}$ and $\Delta^{\Diamond}$, respectively. Briefly `$G'` is related to `$G` by `|L>`, if `$G'` contains exactly those formulas in `$G` which are boxed. Similarly for `$F |R> $F'` only with diamond. The rules are:

```
lstar0   |L>
lstar1   $G |L> $H ==> []P, $G |L> []P, $H
lstar2   $G |L> $H ==>   P, $G |L>      $H
lstar3   [| $G |L> $H; $E |L> $F |] ==>  $E, $G |L>  $F, $H
lstarRF [| $G |L> $H; RF P |] ==>  P, $G |L> P, $H

rstar0   |R>
rstar1   $G |R> $H ==> <>P, $G |R> <>P, $H
```

```
rstar2  $G |R> $H ==>   P, $G |R>      $H
rstar3  [| $G |R> $H; $E |R> $F |] ==>  $E, $G |R>  $F, $H
rstarRF [| $G |R> $H; RF P |] ==>  P, $G |R> P, $H

lstar4  $G |L> $H ==>  $S, $G |L>      $H"
rstar4  $G |R> $H ==>  $S, $G |R>      $H
```

Having extended the syntax we need to add rules for the special recognizers `CF`, `RF` and `RT`:

```
RFchop        "[| RF(A); RF(B) |] ==> RF(A^B)"
RFbox         "RF(A) ==> RF([]A)"
RFdia         "RF(A) ==> RF(<>A)"
```

6.5 TDC

The theory TDC encodes the syntax and rules of Duration Calculus as an extension of the IL theory.

6.5.1 Syntax

First of all we declare `sterm` as the class of state terms and type `so` as the type of state formulas. `sterm` is the third level of terms.

```
classes
  sterm < logic

default
  sterm

types
  so

arities
  so :: sterm
```

On the type `so` we define the syntax:

```
tt            :: so                         ("tt")
ff            :: so                         ("ff")
NOT           :: so => so                   ("NOT _" [40] 40)
AND           :: [so, so] => so             (infixr 35)
OR            :: [so, so] => so             (infixr 30)
```

```
IMPLIES       :: [so, so] => so                 (infixr 25)
IFF           :: [so, so] => so                 (infixr 25)

";="          :: ['a, 'a] => so                  (infixl 50)

so            :: ro => so
```

We could have used the same syntax as for `o` and `ro`, but instead we adopt the spelling used in PVS. The `so` function lifts rigid terms to state terms and is our way of allowing global variables in durations which is necessary to encode typed DC (Section 2.3). We deliberately leave out injectio of o into so as this would break the syntax and allow nested duration terms.

We then define the duration operator `dur` and the usual abbreviations:

```
dur           :: so => real
pr            :: so => o                        ("‘_‘")
point         :: o                              ("‘‘")

len_def        "len == dur(tt)"
pr_def         "pr(s) == dur(s) = len & len > 0"
point_def      "point == len = 0"
```

6.5.2 Proof Rules

The axioms are straightforward:

$$\frac{}{\vdash \int 0 = 0}\ \texttt{DCAX1}$$

```
DCA1  |- dur(ff) = 0
```

$$\frac{}{\vdash \int 1 = l}\ \texttt{DCAX2}$$

```
DCA2  |- dur(tt) = len
```

$$\frac{}{\vdash \int s \geq 0}\ \texttt{DCAX3}$$

```
DCA3  |- dur(A) >= 0
```

$$\frac{}{\vdash \int s_1 + \int s_2 = \int(s_1 \vee s_2) + \int(s_1 \wedge s_2)} \; \texttt{DCAX4}$$

```
DCA4  |- dur(A) + dur(B) = dur(A OR B) + dur(A AND B)
```

$$\frac{}{\vdash (\int s = x) \frown (\int s = y) \Rightarrow \int s = x + y} \; \texttt{DCAX5}$$

```
DCA5  [| RT x; RT y|] ==> |- (dur(A)=x) ^ (dur(A)=y) --> dur(A)=x+y
```

except for the lifting rule which we discussed in Section 2.3:

$$\frac{}{\vdash \int s_1 = \int s_2} \; \texttt{DCAX6} \quad \text{iff } s_1 \Leftrightarrow s_2 \text{ holds in propositional logic}$$

```
DCA6  $G |- $E, []((Oo A)<->(Oo B)), $F ==>
          $G |- $E, dur(A)=dur(B), $F
```

where `Oo` is a function relating state formulas with flexible formulas:

```
Oo              :: so => o                       ("Oo _")
```

The topic of our previous discussion was how to express the DCA6 rule using only one logic. The original formulation uses two:

$$\text{DCA6}: \quad \int P = \int Q \quad \text{if } P \Leftrightarrow Q \text{ holds in propositional logic}$$

The basis for the semantical justification of our formulation of the rule is that $\int A = \int B$ if A is the same function as B (recall that states are functions from time to booleans) ie., by extensionality, if $A(t) = B(t)$ for a t in the context interval. However, in our syntactic encoding states are not functions so we cannot express this premise directly. Instead we use a trick: In an interval $[b, e]$ we interpret the term (OoA) as A at the left interval end-point b. Consequently $(OoA) \Leftrightarrow (OoB)$ expresses that A and B are equal at b and boxing this formulas effectively quantifies over all time points in the context interval. Appendix E contains a semantical encoding of this rule in PVS where its soundness has been proven.

`Oo` is equipped with the rules:

```
Ooeq_o          Oo((x::so) ;= y) == (Oo x) = (Oo y)

Ooso            Oo(so(x)) == no(x)
```

```
Oott          Oo tt == True
Ooff          Oo ff == False

Oonot         Oo(NOT A) == ~(Oo A)
Ooand         Oo(A AND B) == (Oo A) & (Oo B)
Ooor          Oo(A OR B) == (Oo A) | (Oo B)
Ooimplies     Oo(A IMPLIES B) == (Oo A) --> (Oo B)
Ooiff         Oo(A IFF B) == (Oo A) <-> (Oo B)
```

After application of the DCA6 rule, repeated rewriting with the above *O rules* will move the Oo inwards and reveal a logical formula with atomic subformulas of the form (OoP). Propositional reasoning can then be applied on this formula.

The induction rules are encoded as follows:

$$\frac{\Gamma \vdash R(\lceil\rceil), \Delta \qquad \Gamma \vdash (R(X) \Rightarrow R(X \vee \lceil s \rceil \frown X \vee \lceil \neg s \rceil \frown X), \Delta}{\Gamma \vdash R(true), \Delta} \ \texttt{chop\_induct}$$

```
chop_induct  [| $G |- H(''), $D;
               !! X . $G |- H(X) --> H(X|(X^'A')|(X^'NOT A')), $D
             |] ==> $G |- H(True), $D
```

$$\frac{\Gamma \vdash R(\lceil\rceil), \Delta \qquad \Gamma \vdash (R(X) \Rightarrow R(X \vee X \frown \lceil s \rceil \vee X \frown \lceil \neg s \rceil), \Delta}{\Gamma \vdash R(true), \Delta} \ \texttt{induct\_chop}$$

```
induct_chop  [| $G |- H(''), $D;
               !! X . $G |- H(X) --> H(X|('A'^X)|('NOT A'^X)), $D
             |] ==> $G |- H(True), $D
```

For use when rewriting we encode the relation between state predicates and meta-equality:

```
P_reflection  "|- 'A' <-> 'B' ==> (A == B)"
```

6.6 SREAL

The SREAL encodes real valued states. First we declare the sreal type:

```
types
  sreal

arities
  sreal :: sterm
```

and then the syntax:

```
"++"          :: [sreal, sreal] => sreal      (infixl 60)
"--"          :: [sreal, sreal] => sreal      (infixl 60)
"---"         :: sreal => sreal               ("--- _" [80] 80)
"**"          :: [sreal, sreal] => sreal      (infixl 70)

"<<"          :: [sreal, sreal] => so         (infixr 50)
"<<="         :: [sreal, sreal] => so         (infixr 50)
">>"          :: [sreal, sreal] => so         (infixr 50)
">>="         :: [sreal, sreal] => so         (infixr 50)

sr            :: rreal => sreal

Or            :: sreal => real                ("Or _")
```

Notice that we do not define division as it is not closed on the domain of analytic states that we have chosen [40]. The operator spellings are a bit funny to avoid clashes with previously defined syntax.

`Or` is used in application of the DCA6 rule. The rules for `Or` are as expected:

```
Oadd          Or(x ++ y) == (Or x) + (Or y)
Osub          Or(x -- y) == (Or x) - (Or y)
Omult         Or(x ** y) == (Or x) * (Or y)

Olt           Oo(x << y) == (Or x) < (Or y)
Ole           Oo(x <<= y) == (Or x) <= (Or y)
Ogt           Oo(x >> y) == (Or x) > (Or y)
Oge           Oo(x >>= y) == (Or x) >= (Or y)

Oeq_real      Oo((x::sreal) ;= y) == (Or x) = (Or y)
Osr           Or(sr(x)) == nr(x)
```

6.7 EDC

The EDC theory encodes Extended Duration Calculus and extends SREAL which provided the syntax for real valued states. The new operators of EDC are integration and beginning- and end-point value:

```
intg         :: sreal => real
bdot         :: sreal => real                  ("b. _")
edot         :: sreal => real                  ("e. _")
```

CNT is the meta-logical predicate recognizing continuous analytic states (as opposed to piece-wise analytic states):

```
CNT            :: sreal => prop                       ("CNT _")
```

The rules of CNT are:

```
CNTsr          CNT(sr(x))
CNTadd         [| CNT x; CNT y|] ==> CNT(x ++ y)
CNTsub         [| CNT x; CNT y|] ==> CNT(x -- y)
CNTmult        [| CNT x; CNT y|] ==> CNT(x ** y)
```

The rules of EDC are the ones from Ravn [40]:

```
integration [|RT v1; RT v2|] ==>
           |- intg(s)=v1 ^ intg(s)=v2 --> intg(s) = v1+v2

values_b    RT v ==> |- (b. s)=v <-> (b. s)=v ^ True
values_e    RT v ==> |- (e. s)=v <-> True^ (e. s)=v

continuity  [| CNT cs; RT vb; RT ve |] ==>
           |- (e. cs) = ve ^ (b. cs) = vb --> ve=vb

continuity_point CNT cs ==> |- len=0 --> (e. cs) = (b. cs)

mean_inc    [| CNT cs; RT v; RT vb; RT ve |] ==>
           |- (b. cs)=vb & (e. cs)=ve & vb<=v & v<=ve -->
                 (e. cs)=v ^ True
mean_dec    [| CNT cs; RT v; RT vb; RT ve |] ==>
           |- (b. cs)=vb & (e. cs)=ve & ve<=v & v<=vb -->
                 (e. cs)=v ^ True
```

We also add useful rules linking integration with duration. The inclusion of these rules are motivated by the Steam Boiler example.

```
P_le        |- `s<<=sr(v)` --> (b. s)<=nr(v) & (e. s)<=nr(v)
P_ge        |- `s>>=sr(v)` --> (b. s)>=nr(v) & (e. s)>=nr(v)

P_lt        |- `s<<sr(v)` --> ((e. s)<nr(v))^((b. s)<nr(v))
P_gt        |- `s>>sr(v)` --> ((e. s)>nr(v))^((b. s)>nr(v))

b_lt        |- (b. s)<nr(v) & len>0 --> `s<<sr(v)`^True
b_gt        |- (b. s)>nr(v) & len>0 --> `s>>sr(v)`^True
e_lt        |- (e. s)<nr(v) & len>0 --> True^`s<<sr(v)`
```

```
e_gt        |- (e. s)>nr(v) & len>0 --> True^`s>>sr(v)`

P_eq_intg   |- `s;=sr(v)` --> intg(s)=nr(v)*len
P_lt_intg   |- `s<<sr(v)` --> intg(s)<nr(v)*len
P_le_intg   |- `s<<=sr(v)` --> intg(s)<=nr(v)*len
P_gt_intg   |- `s>>sr(v)` --> intg(s)>nr(v)*len
P_ge_intg   |- `s>>=sr(v)` --> intg(s)>=nr(v)*len

point_intg  |- len=0 --> intg(s)=0

mean_gt     [| CNT cs; RT vb; RT ve |] ==>
             |- len>0 & (b. cs)=vb & (e. cs)=ve & vb<=nr(v) & nr(v)<ve
                 --> True^((b. cs)=nr(v) & `cs>>sr(v)`)
mean_lt     [| CNT cs; RT vb; RT ve |] ==>
             |- len>0 & (b. cs)=vb & (e. cs)=ve & vb>=nr(v) & nr(v)>ve
                 --> True^((b. cs)=nr(v) & `cs<<sr(v)`)
```

6.8 FB

The encoding of the standard forms of section 2.5 is simple. The operators are declared and defined by meta-equality. The theorems capturing properties of the operators are all derived rules.

```
fb       :: [o, o] => o          ("_ --->/ _" [26,25] 25)
lt       :: [o, real, o] => o    ("_ -= _ ->/ _" [26,50,25] 25)
ut       :: [o, real, o] => o    ("_ -<= _ ->/ _" [26,50,25] 25)

fb_def   A--->B ==  ALL r. []((A & len=r)^(len>0) --> (len=r)^B^True)
lt_def   A-=t->B == (A & len=t)--->B
ut_def   A-<=t->B == (A & len<=t)--->B
```

6.9 Summary

The encoding of DC in Isabelle consists of a hierarchy of theories. We have presented these theories one by one from the buttom up and described the syntax and rules that they provide. The encoding is the basis from which further rules and theorems may be derived and tactics defined.

Chapter 7

Proof Support

The previous chapter described the encoding of Duration Calculus as a collection of Isabelle theories declaring the basic types and rules of DC. While it is possible to conduct proofs using these rules alone, one quickly realize that such proofs follows recurring patterns and that it is convenient to express these patterns as derived rule or more generally as tactics.

This chapter will describe the most useful of these derived rules and tactics which we have developed during our use of Isabelle/DC. In the presentation we will follow the structure of the previous chapter and present the rules and tactics that each theory give rise to.

Another important aspect of proof support is the use of external decision procedures. We devote the next Chapter to a description of the two decision procedures integrated into Isabelle/DC.

7.1 PL

In this section we describe rules and tactics for the predicate logic theory of `PL`.

7.1.1 TAIL and NTH

Recall that a tactic is a function mapping a proof goal to a sequence (lazy list) of new proof goals.

The following tactics are useful for selecting a particular resolvent when doing resolution. They serve a similar purpose as repeated use of the built-in `back` function, but they work in proof scripts.

```
TAIL : tactic -> int -> tactic
NTH : tactic -> int -> tactic
```

`TAIL` *tac* n is the tactic which returns the n'th tail of the sequence returned by *tac* or the empty sequence if this sequence has fewer than n elements.

`NTH` *tac n* is the tactic which returns the singleton sequence consisting of the n'th element of the sequence returned by *tac* or returns the empty sequence if this element does not exist.

7.1.2 Structural Reasoning

We start by deriving structural rules working on formulas rather than sequences:

```
contR  $H |- $E, P, P, $F ==> $H |- $E, P, $F
contL  $H, P, P, $G |- $E ==> $H, P, $G |- $E
thinR  $H |- $E, $F ==> $H |- $E, P, $F
thinL  $H, $G |- $E ==> $H, P, $G |- $E
exchR  $H |- $E, Q, P, $F ==> $H |- $E, P, Q, $F
exchL  $H, Q, P, $G |- $E ==> $H, P, Q, $G |- $E
```

It is often convenient to be able to bring an formula forward in the sequences of a sequent. This affects the sequence of resolvents in a later resolution. We derive:

```
firstL  P, $H, $G |- $E ==> $H, P, $G |- $E
firstR  $G |- P, $E, $F ==> $G |- $E, P, $F
```

and construct tactics using these rules:

```
val firstL_tac : int -> int -> tactic
val firstR_tac : int -> int -> tactic
```

`firstL_tac` *k i* brings the k'th formula in the antecedent of the i'th subgoal to the front.

`firstR_tac` *k i* brings the k'th formula in the consequent of the i'th subgoal to the front.

Sometimes it is necessary to duplicate a formula in the antecedent or consequent. `dupR_tac` and `dupL_tac` are tactics to do exactly that. They apply the rules `contL` and `contR` and combine with `NTH` to copy a selected formula.

```
dupL_tac : int -> int -> tactic
dupR_tac : int -> int -> tactic
```

`dupL_tac` *k i* will duplicate the k'th antecedent formula of subgoal i.

`dupR_tac` *k i* will duplicate the k'th consequent formula of subgoal i.

7.1.3 Propositional Reasoning

The basic propositional rules are cumbersome to use one by one. Often we want to apply them repeatedly in various ways. To do this we define:

```
flat_tac : int -> tactic
split_tac : int -> tactic
splitnL_tac : int -> int -> tactic
splitnR_tac : int -> int -> tactic
smash_tac : int -> tactic
```

`flat_tac` i does disjunctive simplification on the i'th subgoal. This involves repeatedly transforming antecedent formulas of the form `~`A and A`&`B and consequent formulas of the form A, A`|`B and A`-->`B into a list of formulas. `flat_tac` may prove the subgoal but never produces additional subgoals.

`split_tac` i does conjunctive splitting on the i'th subgoal. This involves repeatedly splitting antecedent formulas of the form A`-->`B and A`|`B and consequent formulas of the form A`&`B into additional subgoals.

`splitnL_tac` n i does conjunctive splitting on the n'th antecedent formula of the i'th subgoal.

`splitnR_tac` n i does conjunctive splitting on the n'th consequent formula of the i'th subgoal.

`smash_tac` i does both disjunctive simplification and conjunctive splitting on the i'th subgoal.

Notice that the above tactics work by matching rather than by resolution. They will not instantiate unknowns in the proof state and thus does not affect provability of the i'th or other subgoals.

7.1.4 Reasoning About Side Conditions

We need a tactic to discard subgoals involving the meta-predicates `RT` and `CF` recognizing rigid and chop free formulas, respectively. Such subgoals arise from the quantifier rules. We define:

```
aside_tac : thm list -> int -> tactic
aside_solve_tac : thm list -> int -> tactic
```

`aside_tac` *aside_rls* i repeatedly applies the rules in *aside_rls* to the i'th subgoal using matching.

`aside_solve_tac` *aside_rls* i behaves like `aside_tac` except it fails if it cannot discard the subgoal.

7.1.5 Quantifier Reasoning

These simple tactics combine the application of the quantifier rules with discarding of side conditions and smashing of resulting subgoals.

```
val allR_tac : int -> tactic
val exL_tac  : int -> tactic
val allL_tac : tactic -> string -> int -> tactic
val exR_tac  : tactic -> string -> int -> tactic
```

`allR_tac` *i* resolves the *i*'th subgoal with the `allR` rule and smash the resulting subgoal.

`exL_tac` *i* resolves the *i*'th subgoal with the `exL` rule and smash the resulting subgoal.

`allL_tac` *aside_solve_tac x i* instantiates the `allLRT_tac` and `allLCF` rules with x, discards any side conditions and smashes the result.

`exR_tac` *aside_solve_tac x i* instantiates the `exRRT_tac` and `exRCF` rules with x, discards any side conditions and smashes the result.

There are similar tactics for the rigid quantifiers:

```
val allrR_tac : int -> tactic
val exrL_tac  : int -> tactic
val allrL_tac : string -> int -> tactic
val exrR_tac  : string -> int -> tactic
```

7.1.6 Lemma Introduction

A lemma is a sequent `|-`A. Lemmas can be introduced by the `lemma` rule using the tactics below.

```
lemma  [| |- A; A, $G |- $E |] ==> $G |- $E

lemma_tac      : thm -> int -> tactic
lemma_inst_tac : (int -> tactic) ->
                 (string*string)list -> thm -> int -> tactic
Lemma_inst_tac : (string*string)list -> thm -> int -> tactic
```

`lemma_tac` *lem i* will introduce the lemma *lem* as the first antecedent formula in subgoal *i*.

`lemma_inst_tac` *tac inst lem i* will introduce the lemma *lem* instantiated with *inst* as the first antecedent formula in subgoal *i*. *tac* is applied to all new subgoals.

`Lemma_inst_tac` *inst lem i* works like `lemma_inst_tac` but will discard side conditions and smash remaining new subgoals.

7.1.7 Simplification

We instantiate the Isabelle simplifier described in section 5.6 to do in-formula propositional simplification. The basis of this instantiation is the reflective rule `iff_reflection`.

We derive a number of equivalences exploiting the identity and zero-elements of the logical operators, their reflectiveness etc. and use them as rewrite rules.

The result is a simplification set to be used with the simplification tactic:

```
val PL_ss : simpset

val simp_tac : simpset -> int -> tactic
```

7.2 RA

The `RA` theory adds real-valued terms to the encoding as flexible terms `real` and rigid terms `rreal`. To convert between the two it further declares the function `nr` and the rules defining it:

```
nr :: rreal => real

nradd   nr(x) + nr(y) == nr(x + y)
nrsub   nr(x) - nr(y) == nr(x - y)
nrneg   --(nr(x)) == nr(--x)
nrmult  nr(x) * nr(y) == nr(x * y)
nrdiv   nr(x) div nr(y) == nr(x div y)
```

It is often convenient to maximize the size of rigid terms. This can be achieved by repeatedly rewriting with these `nr` rules using `rewrite_goals_tac` to move the `nr` up in the term.

The reasoning on real arithmetic will be carried out by a decision procedure described in the next chapter.

7.3 IL

7.3.1 Reasoning About Side Conditions

Side conditions in the `IL` theory involve calculation of $\Gamma^\Box$ and $\Delta^\Diamond$ in the `boxR` and `diaL` rules. The functions are expressed in the encoding as relations (`|L>`, `|R>`) on the argument and the result.

```
boxR  [| $G |L> $G'; $E |R> $E'; $F |R> $F';
        $G' |- $E', P, $F'|]  ==>  $G |- $E, []P, $F
```

The tactic `star_tac` proves `|L>`, `|R>` `|RF>` subgoals using repeated resolution with the `lstar`, `rstar` and `rfstar` rules. The aim of the tactic is to instantiate the primed sequence unknowns `$G'`, `$E'` and `$F'` in the fourth premise.

```
star_tac : tactic -> thm list -> int -> tactic
```

`star_tac` *aside_solve_tac star_rls i* discards the *i*'th subgoal using the rules in *star_rls* as described above. Side conditions are solved using *aside_solve_tac*.

Let us consider how `star_tac` discards a goal `$G |L> $?G`. The rules consider the formulas in the sequence from left to right. From looking at the `lstar` rules it is clear that resolution must be done in the right order if boxed and rigid formulas are to be preserved. If `P` is boxed or if it can be proven rigid (by `aside_solve_tac`) it is kept by the `lstar1` and the `lstarRF` rules, respectively. Otherwise `P` is removed by resolution with the `lstar2` rule. The `lstar3` rule is only applied if the first premise can be solved immediately by assumption. This is often the case when deriving rules.

```
lstar0       "|L>"
lstar1       "$G |L> $H ==> []P, $G |L> []P, $H"
lstar2       "$G |L> $H ==>   P, $G |L>      $H"
lstar3       "[| $G |L> $H; $E |L> $F |] ==>  $E, $G |L>  $F, $H"
lstarRF      "[| $G |L> $H; RF P |] ==>  P, $G |L> P, $H"
```

7.3.2 Chop Chains

Formulas with nested chops occur frequently. We define a subset of such formulas called *chop chains*. These turn out to be very useful when considering monotonicity of chop in the next section.

```
A2  |- (A^B)^C <-> A^B^C
```

Definition: Chop Chain A formula A is a *chop chain* iff it can be rewritten using associativity of chop (`A2`) in to a formula $A_1 \frown \ldots \frown A_k$ for some $k \geq 1$. We use the sequence $A_1, \ldots, A_k$ to denote an arbitrary chop chain for given $k \geq 1$
□

We further define two special forms of chop chains and tactics to rewrite a chop chain in to these forms:

Definition: Top-form For given k and $n < k$ a chop chain $A_1, \ldots, A_k$ is on *top-form* if it has the form $(A_1 \frown \ldots A_n) \frown A_{n+1} \frown \ldots \frown A_k$.
□

Definition: Center-form For given n, $m < k$ and $n < m$ a chop chain $A_1, \ldots, A_k$ is on *center-form* if it has the form $(A_1 \frown \ldots A_n) \frown (A_{n+1} \frown \ldots \frown A_m) \frown \ldots \frown A_k$.
□

```
top_chop : int -> int -> tactic
center_chop : int -> int -> int -> tactic
```

`top_chop` n i brings an antecedent chop chain of subgoal i on top-form.

`center_chop` n m i brings an antecedent chop chain of subgoal i on center-form.

7.3.3 Monotonicity

Monotonicity of chop is most often used for forward reasoning in the antecedent of a subgoal. To support this we derive the rules:

```
mono_chop  [| $H, $G |- $E, [](A-->B); $H, B^C, $G |- $E |]
           ==> $H, A^C, $G |- $E
chop_mono  [| $H, $G |- $E, [](A-->B); $H, C^B, $G |- $E |]
           ==> $H, C^A, $G |- $E"
chop_mono_chop  [| $H, $G |- $E, [](A-->B); $H, C^B^D, $G |- $E |]
                ==> $H, C^A^D, $G |- $E
```

These three rules allow monotonicity reasoning on any part of a chop chain in either top-form or center-form. Notice that we may also derive similar rules for backwards reasoning from the consequent. However, we have found little use for this proof style and its supporting rules.

Often the first premise of the above rules can be proven without the use of the antecedent or other consequent formulas. In such cases rules like the following are useful:

```
mono_chop_bang  [| |- A-->B; $H, B^C, $G |- $E |] ==>
                   $H, A^C, $G |- $E
```

The above rules introduce the formula `B` as an unknown unless it is instantiated during resolution. Monotonicity may also be used between chopped formulas of the antecedent and consequent:

```
monotone  [| $H, $G |- $E, [](A-->A'), $F;
             $H, $G |- $E, [](B-->B'), $F |] ==>
             $H, A^B, $G |- $E, A'^B', $F"

monotone_bang  [| |- A-->A';  |- B-->B' |] ==>
             $H, A^B, $G |- $E, A'^B', $F"
```

These rules does not introduce unknowns which make them useful in proof search.

7.3.4 Interval Length

The following rules are convenient for introducing a measure of interval length into a subformulas.

```
len_chop  (!! x. RT x ==> $H, (len=x & A)^B, $G, x >= nr(zero) |- $E)
          ==> $H, A^B, $G |- $E"
chop_len  (!! x. RT x ==> $H, A^(len=x & B), $G, x >= nr(zero) |- $E)
          ==> $H, A^B, $G |- $E
chop_len_chop  (!! x. RT x
                 ==> $H, A^(len=x & B)^C, $G, x >= nr(zero) |- $E)
               ==> $H, A^B^C, $G |- $E"
```

Notice that the rules propagates to the top level the property that the measure is non-negative. This fact is useful to the real arithmetic decision procedure.

For summation of length of a chopped chain we define:

```
val len_add_tac : tactic -> string -> i -> tactic
```

`len_add_tac` *aside_solve_tac sum i* proves the the accumulated length of a chopped antecedent formula of subgoal *i* is *sum* and inserts this fact as the antecedent formula `len=`*sum*. The proof is recursive on the structure of the chopped chain. *sum* should be an addition with the same structure. Notice therefore that while chop is right associative addition is left associative.

We illustrate the use of `len_add_tac` in a proof:

```
  Level 0
  (len=x & A)^((len=y & B)^(len=z & C))^(len=r & D) |-
   1. (len=x & A)^((len=y & B)^(len=z & C))^(len=r & D) |-

by (len_add_tac aside_solve_tac "x+((y+z)+r)" 1);

  Level 1
  (len=x & A)^((len=y & B)^(len=z & C))^(len=r & D) |-
   1. (len=x & A)^((len=y & B)^(len=z & C))^(len=r & D),
      len=x+((y+z)+r) |-
```

7.3.5 Modal Reasoning

We derive the standard rules for box and diamond modalities:

```
diaL  [| $G |L> $G';  $H |L> $H';  $E |R> $E'; $H', P, $G' |- $E'|]
       ==> $H, <>P, $G |- $E"

diaR  $E |- $F, P, $G  ==>  $E |- $F, <>P, $G
boxL  $E, P, $F |- $G  ==>  $E, []P, $F |- $G"
```

and support the rules by tactics:

```
val boxR_tac : tactic -> int -> tactic
val diaL_tac : tactic -> int -> tactic
```

`boxR_tac` *star_solve_tac i* resolves the *i*'th subgoal with the `boxR` rule and discard the side conditions using *star_solve_tac*.

`diaL_tac` *star_solve_tac i* resolves the *i*'th subgoal with the `diaL` rule and discard the side conditions using *star_solve_tac*.

Very often "banged" versions of `boxR` and `diaL` are convenient:

```
boxR_bang  |- A ==> $G |- $E, []A, $F
diaL_bang  A |- ==> $G, <>A, $G |- $E
```

7.3.6 Simplification

Support of simplification of IL formulas naturally extends the simplification support for PL formulas by adding rewrite rules for `False` propagation and associativity of chop.

```
false_chop |- False^A <-> False
chop_false |- A^False <-> False

A2 |- (A^B)^C <-> A^B^C
```

`False` propagation may prove a subgoal if it propagates to the top of an antecedent. Associativity of chop is used for rewriting mostly to get a nicer unparsing without brackets.

Another very convenient use of simplification is for propagating rigid sub-formulas out of chopped formulas. Propagating rigid formulas to the top level makes them available for reasoning by for example decision procedures. This kind of rewriting is based on the following equivalences:

```
chop_RF_equiv       RF R ==> |- A^(R & B) <-> R & A^B
chop_RF_true_equiv  RF R ==> |- A^R <-> R & A^True
RF_chop_equiv       RF R ==> |- (R & A)^B <-> R & A^B
RF_true_chop_equiv  RF R ==> |- R^B <-> R & True^B
```

To bring the rules into play the rigid formula `R` must be propagated to the extreme left of a conjunctive chain `R & B`. To achieve this we employ the following rules for conjunction:

```
conj_assoc      |- (A & B) & C <-> A & B & C
conj_comm       |- A & B <-> B & A
conj_left_comm  |- A & B & C <-> B & A & C
```

The unique and powerful simplification procedure feature of the Isabelle simplifier is used to apply the `conj_comm` and `conj_left_comm` rules *only* when `A` is flexible and `B` is rigid. Effectively this ensure that a chained conjunction is sorted with the rigid formulas to the left.

We declare a new simplification set adding the above rules and simplification procedures to `PL_ss`.

```
val IL_ss : simpset
```

The simplification tactic `simp_tac` and its siblings apply rewriting to an entire subgoal. Often we need to restrict the scope of the rewriting to a single formula of a sequent or even to a sub-formula of a chopped formula. For this purpose we define the following tactics for simplification in the antecedent. One may define similar tactics for use on the consequent, however we have found no use for such tactics.

```
val simpL_tac          : simpset -> int -> tactic
val simp_chop_tac      : simpset -> int -> tactic
val chop_simp_tac      : simpset -> int -> tactic
val chop_simp_chop_tac : simpset -> int -> tactic
```

`simpL_tac` *ss i* simplifies an antecedent formula of subgoal *i* using the simplification set *ss*.

`simp_chop_tac` *ss i* simplifies the left part of a chopped antecedent formula of subgoal *i* using the simplification set *ss*.

`chop_simp_tac` *ss i* simplifies the right part of a chopped antecedent formula of subgoal *i* using the simplification set *ss*.

`chop_simp_chop_tac` *ss i* simplifies the center part of a chopped antecedent formula of subgoal *i* using the simplification set *ss*.

Disjunction distributes over chop:

```
disj_chop  |- (A|B)^C <-> A^C | B^C
chop_disj  |- A^(B|C) <-> A^B | A^C
```

Uncontrolled distribution of disjunction to the top is not a good idea as disjunctive antecedent formulas causes splitting of subgoals. Sometimes however it is useful in controlled forms.

```
val DNF_ss : simpset
val chop_disj_ss simpset
```

`DNF_ss` is a simplification set used for rewriting into disjunctive normal form and `chop_disj_ss` extends this set with the above rules. Rewriting with `chop_disj_ss` effectively splits a formulas up in to a disjunction of chop chains, the elements of which are conjunctions. This may produce a large disjunction and is best used in a controlled manner by for example `chop_simp_tac`.

A tactic like `chop_simp_tac chop_disj_ss` will leave the right part of a chop formulas as a disjunction. To further distribute this disjunction to the top level we may apply the rules:

```
disj_chopL  [| $H, A^C, $G |- $E; $H, B^C, $G |- $E |]
            ==> $H, (A|B)^C, $G |- $E"
chop_disjL  [| $H, A^B, $G |- $E; $H, A^C, $G |- $E |]
            ==> $H, A^(B|C), $G |- $E"
chop_disj_chopL  [| $H, A^B^D, $G |- $E; $H, A^C^D, $G |- $E |]
                 ==> $H, A^(B|C)^D, $G |- $E

disj_chopR  $G |- $E, A^C, B^C, $F ==> $G |- $E, (A|B)^C, $F
chop_disjR  $G |- $E, A^B, A^C, $F ==> $G |- $E, A^(B|C), $F
chop_disj_chopR
            $G |- $E, A^B^D, A^C^D, $F ==> $G |- $E, A^(B|C)^D, $F
```

Notice that we cannot distribute to the top level using rewriting as this would possibly affect subformulas which we want to be unaffected.

7.4 TDC

7.4.1 Equality of Durations

Reasoning about equality of durations can be done by the `DCA6` rule lifting state formulas to the DC formula level. The use of the rule is supported by the `DCA6_tac` tactic.

```
 DCA6  $G |- $E, []((Oo A)<->(Oo B)), $F ==>
          $G |- $E, dur(A)=dur(B), $F

val DCA6_tac : tactic -> thm list -> int -> tactic
```

`DCA6_tac` *O_rls* *i* applies the `DCA6` rule on subgoal *i*. The tactics also removes the box rewriting is performed on the fourth premise using the rules in *O_rls*.

Other useful rules derived from `DCA6` are:

```
DCA6_bang  |- (Oo A)<->(Oo B)  ==> $G |- $E, dur(A)=dur(B), $F

P_monotone_bang  |- (Oo P)-->(Oo Q)  ==> $H, `P`, $G |- $E, `Q`, $F
```

7.4.2 Duration Measures

The following rules are convenient for introducing a measure of a duration in a subformula. The rules and tactics are similar the to ones for length introduction.

```
dur_chop  (!! x. RT x ==> $H,(dur(P)=x & A)^B,$G,x >= nr(zero) |- $E)
          ==> $H, A^B, $G |- $E"
chop_dur  (!! x. RT x ==> $H,A^(dur(P)=x & B),$G,x >= nr(zero) |- $E)
          ==> $H, A^B, $G |- $E
chop_dur_chop  (!! x. RT x
                  ==> $H, A^(dur(P)=x & B)^C, $G, x >= nr(zero) |- $E)
               ==> $H, A^B^C, $G |- $E"
```

For summation of duration measures in a chopped chain we define:

```
val dur_add_tac : tactic -> string -> string -> i -> tactic
```

`dur_add_tac` *aside_solve_tac P sum i* proves that the accumulated duration of a state in a chopped antecedent formula of subgoal *i* is *sum* and inserts this fact as the antecedent formula `dur(P)=`*sum*. The proof is recursive on the structure of the chopped chain. *sum* should be an addition with the same structure.

7.4.3 Simplification

We instantiate the Isabelle simplifier to do propositional simplification of state formulas. The basis of this instantiation is the reflective rule `P_reflection`.

The result is a simplification set to be used with the simplification tactic:

```
val P_ss : simpset
```

7.4.4 P Chains

A *P-chain* is a chop chain of the form 'P_1', ..., 'P_k'. Formulas built from point and state predicates and combined with conjunction, disjunction, chop, diamond can be rewritten into a disjunction of P-chains. This is often a useful simplification because monotonicity applies easily to any subsequence of a chop chain and disjunction distributes over chop.

The rules used for this kind of rewriting are listed below. They are collected in a simpset:

```
val chain_ss : simpset
```

First a number of equivalences which are useful for simplification in general.

```
P_chop       |- `P`^`P` <-> `P`
P_chop_A     |- `P`^`P`^A <-> `P`^A
P_conj       |- `P` & `Q` <-> `P AND Q`
point_chop   |- ``^A <-> A
chop_point   |- A^`` <-> A
```

False distributes over chop and is the identity of disjunction. The following rules help to reduce the number of disjuncted P chains.

```
ff_False     |- `ff` <-> False
point_P      |- `` & `P` <-> False
P_point      |- `P` & `` <-> False
point_chain  |- `` & `P`^A <-> False
chain_point  |- `P`^A & ``<-> False
```

The rules below are used to combine two conjoined P-chains into one P-chain.

```
True_tt      |- True <-> `` | `tt`
chain_P      |- `Q`^A & `P` <-> `P AND Q`^((`P`|``) & A)
chain_chain  |- `P`^A & `Q`^B <->
                `P AND Q`^(A & B) |
                `P AND Q`^(B & `P`^A) |
                `P AND Q`^(A & `Q`^B)
```

7.5 Standard Forms

We prove the theorems for the derived operators listed in Section 2.5. Of these `fb_neg` is very frequently used so we support it with a convenient tactic:

```
fb_neg  |- ~ (A ---> `P`) <-> <>(A ^ `NOT P`)

val fb_neg_tac : int -> tactic
```

`fb_neg_tac` i applies `fb_neg` to the first followed by, leads to or up to formula in the i'th subgoal.

7.6 Extended DC

Two basic theorems of EDC are:

```
exists_b        |- EX x. (b. s) = x
exists_e        |- EX x. (e. s) = x
```

introducing measures for beginning- and end-point values of states. We support these through rules like:

```
b_chop   (!!x. RT x ==> $H, (b. s = x & A) ^ B, $G |- $E)
           ==> $H, A ^ B, $G |- $E
```

and accompanying tactics:

```
val chop_b_tac : string -> int -> tactic
val b_chop_tac : string -> int -> tactic
val chop_b_chop_tac : string -> int -> tactic
val chop_e_tac : string -> int -> tactic
val e_chop_tac : string -> int -> tactic
val chop_e_chop_tac : string -> int -> tactic
```

`chop_b_tac` s i applies rule `b_chop` to the i'th subgoal to introduce a measure for `(b.` s`)` in the left part of a chopped antecedent formula.

7.7 Theorem Collection

We start a collection of (hopefully) useful theorems of DC and its underlying logic by searching a few literature references for theorems and subsequently prove them in Isabelle/DC.

7.7.1 Hughes and Cresswell

These theorems of S4 are form [26] and can also be found in the Isabelle distribution. We use our own naming.

```
HC1   |- []A --> A
HC2   |- []A --> [][]A
HC3   |- []A --> <>A
HC4   |- <>[](<>A --> []<>A)
HC5   |- <>[](<>[]A --> []A)
HC6   |- []P <-> [][]P
HC7   |- <>P <-> <><>P
HC8   |- <>[]<>P --> <>P
HC9   |- []<>P <-> []<>[]<>P
HC10  |- <>[]P <-> <>[]<>[]P
HC11  |- []P | []Q <-> []([]P | []Q)
HC12  |- ((P>-<Q) --< R) --> ((P>-<Q) --< []R)
```

7.7.2 Hailpern

These theorems of S4 are from [19] and can also be found in the Isabelle distribution. We use our own naming.

```
H1   |- [](P & Q) <-> []P & []Q
H2   |- <>(P | Q) <-> <>P | <>Q
H3   |- <>(P --> Q) <-> ([]P --> <>Q)

H4   |- [](P --> Q) --> (<>P --> <>Q)
H5   |- []P --> []<>P
H6   |- <>[]P --> <>P

H7   |- []P | []Q --> [](P | Q)
H8   |- <>(P & Q) --> <>P & <>Q
H9   |- [](P | Q) --> []P | <>Q
H10  |- <>P & []Q --> <>(P & Q)
H11  |- [](P | Q) --> <>P | []Q
```

7.7.3 Zhou and Hansen

These theorems of IL and DC are from [8].

Theorems of IL

```
IL1    |- [](A-->B) --> ([]A --> []B)

IL2    |- []A --> A

IL3    |- []A --> [][]A

IL4    |- A ==> |- []A

IL5    |- []A --> ~((~A)^B)

IL6    |- []A --> B ==> |-[]A --> []B

IL7    |- [](A-->A') --> (A^B --> A'^B)

IL8    |- [](A-->B) --> []([]A --> []B)

IL9    RF(A) ==> |- A --> []A

IL10a  [| RF(A(x)); |- (ALL x. A(x)-->B(x)^C(x))|] ==>
```

```
            |- (ALL x. A(x)-->B(x))
IL10b  [| RF(A(x)); |- (ALL x. A(x)-->B(x)^C(x))|] ==>
            |- (ALL x. A(x)-->C(x))

IL11   |- All x. len=x --> A ==> |- A

IL12   |- (EX x. A(x)^B(x)) --> (EX x.A(x))^(EX x.B(x))

IL13a  |- A^False <-> False
IL13b  |- False^A <-> False

IL14a  |- (A|B)^C <-> A^C | B^C
IL14b  |- A^(B|C) <-> A^B | A^C

IL15a  RT x ==> |- (len=x & A)^B --> ~((len=x & ~A)^B)
IL15b  RT x ==> |- A^(len=x & B) --> ~(A^(len=x & ~B))

IL16a  |- (A & len=x)^B & (A' & len=x)^B' -->
              (A & A' & len=x)^(B & B')
IL16b  |- A^(B & len=x) & A'^(B' & len=x) -->
              (A & A')^(B & B' & len=x)

IL17a  |- A^(len=nr(zero)) --> A
IL17b  |- (len=nr(zero))^A --> A

IL18a  |- []A & B^C --> (A & B)^C
IL18b  |- []A & B^C --> B^(A & C)

IL19a  |- (len>nr(zero))^(len>nr(zero)) --> len>nr(zero)
IL19b  |- (len>=nr(zero))^(len>nr(zero)) --> len>nr(zero)
IL19c  |- (len>nr(zero))^(len>=nr(zero)) --> len>nr(zero)

IL20   |- [](A & B) <-> []A & []B

IL21   |- []A | []B --> [](A | B)
```

Theorems of DC

```
DC1    |- `` | True^`P` | True^`NOT P`

DC2    |- `` | `P`^True | `NOT P`^True

DC3    |- dur(P) + dur(NOT P) = len
```

```
DC4     |- dur(P) <= len

DC5     |- dur(P) >= dur(P AND Q)

DC10    |- ‘P‘ & ‘Q‘ --> ‘P AND Q‘

DC11    |- ‘P AND Q‘ --> ‘P‘

DC18    |- ‘P‘^True^‘NOT P‘ --> ‘P‘^‘NOT P‘^True

DC19    |- ‘‘ |
           True^‘NOT P AND NOT Q‘ |
           True^‘P‘ |
           True^‘NOT P AND NOT Q‘^‘P‘ |
           True^‘Q‘^‘P‘ |
           True^‘Q‘
```

DC1, DC2, DC10, DC11, DC18 and DC19 are easily proved using the DC-DECIDE tactic. Proving DC3 and DC4 involves the lifting rule DCA6. DC1 has also be proved using the induction rule chop_induct.

7.8 Summary

Derived rules and tactics for their application are central to providing proof support for a logic. In this chapter we presented the proof support for DC which we have developed. We structured our presentation around the theory hierarchy of the encoding and described proof support for individual theories.

Chapter 8

Decision Procedures

This chapter will present the decision procedures used in Isabelle/DC and how they are integrated in to the tool. The procedures give automatic reasoning support for a subset of DC and for linear real arithmetic.

We start by describing the *oracle* facility of Isabelle for integrating such procedures and how such a facility may affect the integrity of Isabelle proofs. Then we describe the procedures and their integration into Isabelle and argue that this integration has no adverse affect on the integrity and soundness of Isabelle.

Our use of oracles adds two new theories `RAOracle` and `DCOracle` into the theory hierarchy presented in Chapter 6. The sole purpose of the new theories is to declare the oracles. The augmented hierarchy appears in figure 8.1.

8.1 Oracles

Oracle is the Isabelle name for an external decision procedure or theorem prover. What an oracle can do is basically to inject formulas into Isabelle as theorems.

Recall that internally an Isabelle theorem is a datatype of the type `thm` and that the operations supported and allowed by Isabelle on this structure are exactly functions corresponding to the sound primitive rules of the meta logic. This is what ensures the integrity of Isabelle, that only formulas that are valid can be derived as theorems. Injecting theorems from an external source may jeopardize soundness.

In short an oracle is an SML function that when given a problem to decide (eg. a formula) will either return a formula if it succeeds or will raise an exception if it fails. The formula is subsequently type-checked and turned into a theorem by Isabelle. This is where soundness may be jeopardized if in fact the formula is invalid The oracle function may decide the problem by calling external programs or by other means.

Different oracles may depend on different information for their operation. A trick is used to allow for this diversity when typing oracle functions in the strict type system of SML. The information is represented by an exception value of type `exn`. In SML the exception type `exn` is a datatype that may be extended with new constructors any time. This is

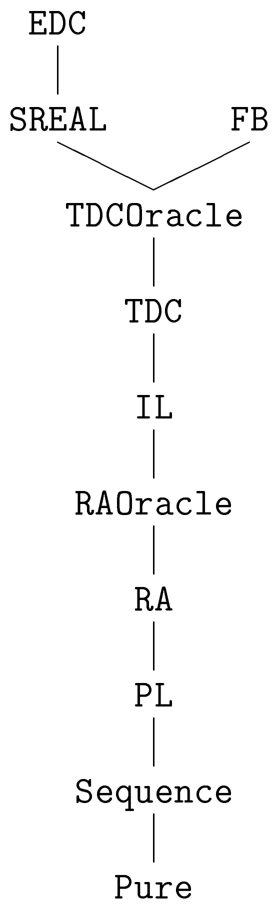


Figure 8.1: Hierarchy of Isabelle/DC theories including oracles

exploited by each oracle having their own exception constructor carrying the information they need.

Thus formally an oracle is a function of type `Sign.sg * exn -> term` where `Sign.sg` is the SML type of a theory signature. The `Sign.sg` type represents information that allows the oracle to distinguish variables from constants etc.

A theory may be associated with one oracle. This is done via the `set_oracle` function:

```
add_oracle    : bstring -> (Sign.sg * exn -> term) -> theory -> theory
```

Invoking an oracle on a problem constructs a theorem (or raises an exception). The theory identifies the oracle to invoke.

```
invoke_oracle : theory * xstring * Sign.sg * exn -> thm
```

8.2 DC-DECIDE

DC-DECIDE is a decision procedure for a subset set of DC called *Decidable Duration Calculus* (DDC) and described in [45]. The procedure gets its name from the proof command in PC/DC that invokes it. Basically the subset includes DC formulas built from state predicates and combined with the modalities and the propositional connectives. The subset is defined via the syntax:

$$
\begin{array}{rcll}
P, Q & ::= & 0 \mid 1 \mid s \mid \neg P \mid P \vee Q & \text{State formulas} \\
A, B & ::= & \lceil P \rceil \mid \neg A \mid A \vee B \mid A \frown B & \text{Formulas}
\end{array}
$$

with symbols as defined in Chapter 2. Besides the usual abbreviations for the logical connectives and modalities the following abbreviations are used:

$$
\begin{array}{rcl}
\mathit{false} & \hat{=} & \lceil 0 \rceil \\
\mathit{true} & \hat{=} & \neg \mathit{false} \\
\lceil\,\rceil & \hat{=} & \neg \lceil 1 \rceil
\end{array}
$$

8.2.1 Checking Satisfiability

The core procedure of DC-DECIDE checks satisfiability of a DDC formula. Validity is expressible as non-satisfiability of the negated formula.

Recall from Chapter 2 that a DC formula A is satisfiable if an interpretation $\mathcal{I}$ exists such that A is true for that interpretation.

DDC formulas are limited in that they only express temporal properties and do not mention interval length. Semantically this means that we can make do with a less fine grained domain than the domain of interpretations. Instead we use the domain of *behaviours*.

Let A be a DDC formula, S be the set of state variables occurring in A and PS be the set of subsets of S. A behaviour $\mathcal{H}$ of A is a (possibly infinite) sequence $a_1, a_2, \ldots, a_i, \ldots$ of elements in PS. Each a_i denote a state where a state variable s is true iff $s \in a_i$.

The idea behind the procedure is to exploit that the set of behaviours satisfying A can be characterized precisely by a regular language $R(A)$ over the alphabet PS. The details of this can be found in [45]. Non-satisfiability of A then becomes the question of checking emptyness of the language $R(A)$.

It is well-known that checking emptyness of a regular language is decidable. Furthermore the construction of $R(A)$ is computable so the combined procedure for checking satisfiability is also decidable. The complexity, however, is non-elementary.

8.2.2 Interface

The procedure was implemented in ML by Peter Sestoft and Jens Ulrik Skakkebæk and the source code is available. The interface of the procedure is as stated below:

```
type Formula

datatype result  =
    Valid
  | Unsatisfiable
  | Counter_example of Formula
val dense_valid: Formula -> result
```

The type `Formula` represents the abstract syntax of DDC formulas. `dense_valid` is the decision procedure itself. It can give three answers represented as the type `result`.

8.2.3 Oracle

We build the oracle on top of the interface described above. We aim for simplicity to easy the argument of soundness. First we convert the result returned by `dense_valid` into a formula with the function:

```
 dense_valid_fmla : Formula -> Formula
```

`dense_valid_fmla` F returns a valid formula constructed from the result of `dense_valid` F. If F is valid then return F else if F is unsatisfiable then return $\neg F$ else F has a counter example E so return $E \Rightarrow \neg F$.

Clearly the formula returned by `dense_valid_fmla` is valid. What is missing is a means of converting between DDC formulas and Isabelle/DC formulas. For this we construct:

```
val term_of : Formula -> term
```

`term_of` F converts F to an equivalent term of type `o`. State names are converted to free variables.

The `term_of` function is a simple recursive function traversing F and construction an equivalent term. It is straightforward to check that this conversion is correct. Now we can construct the oracle:

```
exception DCOracleExn of Formula

val TDC_oracle : Sign.sg * exn -> term
```

`TDC_oracle` *sign* (`TDCOracleExn` F) returns the sequent `|-` A where A is the conversion of the result of `dense_valid_fmla` F.

We conclude that the oracle is sound assuming that the procedure implemented by Sestoft and Skakkebæk is correct.

8.2.4 Tactic

It turns out to be more involved to construct a useful tactic to employ the oracle than to construct the oracle itself. Fortunately we can relax somewhat in this work because the potential danger of introducing unsoundness has been isolated in the oracle.

Our aim is to construct a tactic to discard a subgoal using the DC oracle. There are two problems in this: Firstly to extract from the subgoal a DDC formula to hand to the oracle and secondly to instantiate the theorem returned by the oracle and use the instantiation to discard the subgoal.

Since the oracle will only accept DDC formulas we need to extract from the sequent a DDC formula which is a sufficient premise of the sequent i.e. is strong enough that we can prove the sequent from it. On the other hand of course the extracted formula must be weak enough to actually be valid. We define:

```
val extract_sufficient_DDC_premise :
        term -> (string*string) list * Formula
```

`extract_sufficient_DDC_premise` *subgoal* extracts a DDC formula F from *subgoal* and returns $(insts, F)$. F is suitable for handing to the DC oracle and *insts* are instantiations for the theorem returned by the oracle necessary to discard the subgoal.

Now we can define the tactic:

```
val decide_tac : int -> tactic
```

`decide_tac` i will discard subgoal i using the DC oracle.

and illustrated its use. Consider the subgoal:

```
('R=sr(x)' & A)^'R=sr(x)', B |- 'R=sr(x)', C
```

A sufficient DDC premise F of this goal is `'P'^'P' --> 'P'` with the instantiations *insts* = `[("P", "R=sr(x)")]`. Clearly F is valid and when instantiated with *insts* it becomes (`{'R=sr(x)'^'R=sr(x)' --> 'R=sr(x)'`) and can be used to prove the subgoal.

8.3 SVC

SVC is short for Stanford Validity Checker, a tool combining decision procedures for a number of theories among these quantifier-free linear real arithmetic.

8.3.1 Interface

SVC is distributed as a C library and as a UNIX command line tool with an ASCII syntax for formulas communicated through input and output files. We construct an SML interface is to the command line tool consisting of the structure Svc with signature:

```
structure Svc :
  sig
    datatype Expr = ...

    val valid : Expr -> bool
  end
```

`Expr` is an abstract syntax for the formulas accepted by SVC.

`valid` *e* decides validity of the SVC formula *e* by unparsing the abstract syntax to the ASCII syntax accepted by the command line tool, calling the tool and extracting the answer from the output.

8.3.2 Oracle

```
val Expr_of : term -> Expr

exception RAOracleExn of term

val RAOracle : Sign.sg * exn -> term
```

The oracle builds on this interface and essentially supplies a function `Expr_of` to translate Isabelle formulas to the SVC abstract syntax `SVC`. The function of the oracle is really quite simple. If the translated formulas is found to be valid by SVC then the formula (the untranslated one) is returned from the oracle in the form of a sequent which is turned in to a theorem by Isabelle, otherwise an exception is raise.

The `Expr_of` deserves a little more attention. SVC allows rational constants but has no division operator. This means that a division a/b with constant integer expression b are still translatable as $a * 1/b$ where $1/b$ is a rational number.

The soundness of the oracle depends on the SVC tool and the `valid` and `Expr_of` functions. It is not hard to check that the two functions are correct.

8.3.3 Tactic

The problems in constructing a tactic employing the RA oracle are much the same as for the `decide_tac` tactic for DDC reasoning.

We define:

```
val extract_sufficient_SVC_premise :
       term -> (string*string) list * Expr
```

`extract_sufficient_SVC_premise` *subgoal* extracts an SVC formula F from *subgoal* and returns $(insts, F)$. F is suitable for handing to the RA oracle and *insts* are instantiations for the theorem returned by the oracle necessary to discard the subgoal.

and the tactic:

```
val ra_tac : thm list -> int -> tactic
```

`ra_tac` *RA_rls* i will rewrite the i'th subgoal usign the rules in *RA_rls* and discard the subgoal using the RA oracle.

Rewriting prior to applying the oracle is convenient for expanding definitions like:

```
‘‘ == len=zero
‘P‘ == dur(P) = len & len > zero
len == dur(tt)
```

8.4 Summary

Efficient and powerful decision procedure are key elements of good proof support. In this chapter we described our use of the Isabelle oracle facility to integrate two decision procedures in to our tool. We did this with an emphasis on the soundness of such an integration.

Chapter 9

Steam Boiler Revisited

In this chapter we put our newly developed tool to the test on the Steam Boiler case study described in Chapter 4. We will encode the example in Isabelle/DC and perform the verification using the tool.

The reason for redoing the verification is to obtain data which we can use to compare the proof support provided by PC/DC and Isabelle/DC. We will analyze the Isabelle/DC proofs the same way as in Chapter 4. This gives us a measure to quantify the improvement in proof support provided by Isabelle/DC, if any.

9.1 Encoding

The encoding of the Steam Boiler example in Isabelle/DC is not much different from the one used in PC/DC. The encoding is distributed on four theories related as in Figure 9.1. SB_DEFS defines auxiliary functions, SB_TLT contains the TLT program, SB defines the timing of the program while SB_COMMIT5 defines the assumptions under which the fifth commitment is proved. The details of the encoding can be found in Appendix B.

```
SB_COMMIT5
    |
    SB
    |
  SB_TLT
    |
  SB_DEFS
```

Figure 9.1: Hierarchy of Steam Boiler theories

9.2 Verification

The theorems and lemmas we prove are all collected in Appendix D. Here we will briefly present some useful tactics which we constructed during the verification work. Halfway through the proofs we considered what could be done to make the proofs shorter. It appeared that combining rules for monotonicity and lemma introduction with simplification, moderate proof splitting and application of the decision procedures might be helpful.

The tactics we constructed are the following:

```
val Solver_tac         : int -> tactic

val Monotone_tac       : int -> tactic
val Mono_chop_tac      : string -> int -> tactic
val Chop_mono_tac      : string -> int -> tactic
val Chop_mono_chop_tac : string -> int -> tactic
val Lemma_tac          : thm -> int -> tactic
val Lemma_inst_tac     : (string*string) list -> thm -> int -> tactic
```

`Solver_tac` i combines the application of different decision procedures on the i'th subgoal. It fails if the subgoal is not proven. The tactic tries to prove the goal by application of `ra_tac` possibly after using the rule `P_monotone`. If this fails it tries `decide_tac`.

The remaining tactics all use `Solver_tac` on any new subgoals. By first they do simplification and extraction of rigid sub-formulas from chopped formulas. `Lemma_tac` and `Lemma_inst_tac` also split the introduced lemma. While this may lead to a proliferation of subgoals it turns out that these are most often discarded by `Solver_tac`.

9.3 Comparing Isabelle/DC to PC/DC

For our comparison for Isabelle/DC and PC/DC we group the Isabelle/DC tactics in the same groups as was done with the proof commands of PC/DC (see figure 9.3). Before presenting and discussing the measurements we shortly describe how they are obtained .

9.3.1 How to Measure

An Isabelle proof script is a list of tactics. We analyze the ML source code for these scripts to extract the data we want. The tools we use for this are standard Unix text processing tools like `grep`, `sed` and `wc`. We loose some accuracy in our analysis with this simplistic approach, but it is not substantial. More precise measurements would be possible on the basis of an abstract syntax representation of the script, as we used to analyze the PC/DC proofs. We might also have instrumented each tactic to collect the necessary data, but this approach would mean substantial alterations to tactics.

Group id	Description
prop	Propositional rules
struct	Structural rules
equal	Equality rules, definition expansion
quant	Quantifier rules
rewrite	Rewrite and auto-rewrite rules
mono	DC monotonicity of chop
decide	DC decision procedure call
assoc	DC associativity of chop
modal	DC modality rules for box and diamond
dccase	DC case split
false	DC FALSE propagation and chop
dcquant	DC quantifier rules
propdur	DC duration properties
free	DC Context free terms and chop
edc	EDC rules for b. and e.
disj	DC disjunction and chop
sblem	Steam Boiler lemmas
sbax	Steam Boiler axioms and definitions
math	Lemma about continuous states
other	Ungrouped commands

Figure 9.2: Proof command groups.

9.3.2 Measurements

Here we present our measurements in the form of Figure 9.3 and discuss them in the subsequent section. In the figure g range over the groups in the first column and P_g and I_g are the number of proof commands of group g in the PC/DC and Isabelle/DC proofs, respectively. Column I_g/P_g show the relative change in absolute numbers of group g. Columns p_g and i_g show the distribution of commands on groups for the PC/DC and Isabelle/DC proofs, respectively. The last column (i_g/p_g) shows the relative change in this distribution.

9.3.3 Comparison

First of all we see an overall reduction in the number of proof commands necessary to complete the verification in Isabelle/DC. The reduction is by a factor of around two ($I_{total}/P_{total} = 52.3\%$). Reducing proof size also reduces the time spent on the interactive process of entering proof commands, correcting typos and maintaining proofs when specification or tactics change. In this light a reduction of a factor two is considerable although we should mention that we are still far from reaching the number of proof steps

Group g	P_g	I_g	I_g/P_g	p_g	i_g	i_g/p_g
			%	%	%	%
prop	569	187	32.8	17.0	10.7	62.9
struct	263	71	26.9	7.9	4.0	50.6
equal	180	188	104.4	5.4	10.7	198.1
quant	140	173	123.5	4.2	9.9	235.7
rewrite	85	16	18.8	2.5	0.9	36.0
mono	481	329	68.3	14.4	18.8	130.5
decide	262	67	25.5	7.8	3.8	48.7
assoc	227	112	49.3	6.8	6.4	94.1
modal	174	48	27.5	5.2	2.7	51.9
case	121	96	79.3	3.6	5.5	152.7
false	86	0	0.0	2.5	0.0	0.0
dcquant	72	43	59.7	2.1	2.4	114.2
propdur	72	9	12.5	2.1	0.5	23.8
free	32	9	28.1	0.9	0.5	55.5
edc	26	30	115.3	0.7	1.7	242.8
disj	9	16	177.7	0.2	0.9	450.0
sblem	126	103	81.7	3.7	5.9	159.4
sbax	46	68	147.8	1.3	3.9	300.0
math	28	5	17.8	0.8	0.2	25.0
other	330	173	52.4	9.9	9.9	100.0
total	3329	1743	52.3	100.0	100.0	100.0

Figure 9.3: Proof command counts distributed on groups. P_g = count for PC/DC proof, I_g = count for Isabelle/DC proof, $p_g = P_g/P_{total}$, $i_g = I_g/I_{total}$

one would consider necessary in a convincing pen and paper proof.

Is the reduction even among the groups or has the distribution changed in another way? The extra proof support we have added for the case study mainly target trivial steps like propositional simplification and decision procedure invocation and not higher level steps, which decide how the proof progresses like monotonicity rules, case splits and quantifier instantiation. Therefore we should expect a larger decrease in trivial steps than in higher level steps.

To investigate if this is indeed the case, we split the commands groups into two: The groups containing *trivial* commands and the groups containing *controlling* commands.

How we distribute the groups can be seen in Figure 9.4. We choose to put the *assoc* group among the controlling commands. This group contains the tactics `top_chop_tac` and `center_chop_tac` which are used to rewrite a chop-chain prior the application of a monotonicity rule. In PC/DC this effect can be achieved through optional parameters to the monotonicity commands. We also put lemma application groups *sblem* and *sbax* among the controlling groups as lemmas require proper instantiation like quantifier reasoning.

Controlling	Trivial
quant	*prop*
mono	*struct*
assoc	*equal*
modal	*rewrite*
case	*decide*
sblem	*false*
sbax	*dcquant*
	propdur
	free
	edc
	disj
	math
	other

Figure 9.4: Groups with controlling and trivial commands.

The i_g/p_g column of Figure 9.3 will give the answer to the question. Except for the *edc* and *disj* groups we see a relative decrease in the use of trivial commands. The *edc* show a small absolute increase in usage while the *disj* group show a large relative increase. This is cause by a deliberate decision in Isabelle/DC to not automatically distribute disjunction over chop. This gives more control over proof splitting. The controlling commands on the other hand are more emphasized in the Isabelle/DC proof as was expected. Adding the numbers we see that where controlling commands constituted 39.5% of the commands in the PC/DC proof this number has risen to 53.2% in the Isabelle/DC proof. analyze

We conclude that we have indeed improved the proof support by the Isabelle/DC tool.

Going further will mean targeting the controlling steps and do proof search. Adding more decision procedures for DC subsets will also help.

Chapter 10

Conclusion

In this last chapter we will look back at the previous chapters and the work documented there to summarize what have been done and with the benefit of hindsight and the larger context comment on alternatives for the approaches taken and list subjects of future work.

10.1 Summary

We have developed a theorem prover for Duration Calculus based on the generic proof tool Isabelle. Our starting point has been [21] defining classical Duration Calculus in a Hilbert style and we have strived to always relate our work back to this foundation. Our experience with PC/DC has convinced us of the benefits of a sequent style logic over the Hilbert style for actual theorem proving work. Therefore we have developed a sequent style version of DC and documented its properties as both sound and equally powerful as the Hilbert style version. With a view towards applications we have seen the need to generalize classical DC and introduce typed states in the style of Ravn [40] and the extra operators of Extended DC for expressing properties of continuous states.

PC/DC is an existing theorem prover for DC developed by Skakkebæk [45] and based on PVS [31]. We have worked with this tool to verify a specification of the Steam Boiler by Rischel et al. [42]. We document the experiences of this work and conclude that there is a need for more powerful proof commands than found in PC/DC to ease the burden of the user.

Furthermore our work on supporting and further developing PC/DC has lead us to look for a alternative platform for a DC theorem prover. PVS is a powerful tool for work in its basic logic but lacks (and was never meant to have) features for extension. It is possible to hack such features and the original version of PC/DC does this with success. Still, the closed and undocumented nature of PVS makes it hard to provide maintenance and further development of PC/DC in an environment of ongoing PVS development and release.

Isabelle is the alternative platform we choose for implementation of our proof tool. We document the basic logic of Isabelle and the features it provides for extension. This gives us a basis for presenting the encoding of the sequent style DC developed previously and again we strive to document the relation of the encoding to this formulation of DC and ultimately to the definition by Hansen and Zhou. We do this to ensure the soundness of our encoding and the tool.

We document our work on providing proof support for DC. This is a collection of derived rules and lemmas which have proved themselves useful. Candidates for such a collection reveal themselves during proof work. We benefit from the experiences of ourselves and others and include theorems from the literature as well as rules and lemmas found in PC/DC. We also make use of the proof programming facilities of Isabelle and document tactics and tacticals we have developed for proof search and high level proof steps.

Because such a collection of rules and lemmas are build from experience considerable time goes into collecting it. But it is also worth noting that maintenance of the collection is also time consuming. Proofs of rules and lemmas seldom are resilient to changes in the encoding or the tactics they employ. Experimenting with the encoding or tactics thus comes with a cost.

Our goal is to provide proof support equaling the support found in PC/DC. In this work we find that the open and documented nature of Isabelle ease the work considerably. We also find that our familiarity with ML on which Isabelle is based speeds up this work compared to the closed and LISP based PVS.

Now we revisit the Steam Boiler case study. This time equipped with a proof tool that we can extend easily to provide proof support suitable for this example and hopefully also generally. We document the encoding of the Boiler specification and the tactics we develop. Having two version of the same proofs done with different tools we naturally compare them and find that the Isabelle proofs are generally shorter by a factor of two. We take this as a sign of improved proof support by the Isabelle/DC tool.

10.2 Discussion

In the process of developing the tool we have presented in the previous chapters we have made a number of choices knowingly or unknowingly. With the development work and the case study behind us it is time to re-iterate the rationale behind the choices made and to discuss alternative approaches and points of potential improvement. We devote a section for each subject of our discussion.

10.2.1 Why Sequent Style?

Our experience with the sequent style version of DC it that it worked well in PC/DC for actual deduction work. This is our primary reason for also choosing a sequent style formulation for Isabelle/DC. The system by Skakkebæk also gave us a good starting point for our own formulation and allowed us to concentrate on providing a link to the Hilbert system and to consider the question of a minimal set of rules and axioms, subjects not treated in [45].

One might argue that Isabelle is best tailored for the Natural Deduction style. Its base logic use this style and it is certainly true that the majority of work using Isabelle is with object logics following this style [35]. So why not use the Natural Deduction style for Isabelle/DC?. The answer lies in a lack of a given starting point, a Natural Deduction formulation of DC. A search for such a formulation was seen as a distraction from the main goal of improving proof support for DC.

Still, a Natural Deduction style version of DC is worth investigating. A starting point might be the treatment by Bull and Segerberg [7] of the general question of Natural Deduction in modal logics. Bull and Segerberg propose a natural Deduction formulation of the modal logic **K** and treat other modal logics as axiomatic extensions of this system. Hawthorne [23] extends the system to provide Natural Deduction style versions of the modal logics **T**, **D**, **B**, **S4** and **S5**. IL is an extension of **S4** [21]. Whether these formulation are suitable for encoding in Isabelle is an open question.

10.2.2 Why Isabelle and not PVS?

We have touched upon this question before. The primary reason for abandoning PC/DC and the underlying PVS was the closed and undocumented nature of PVS which made PC/DC hard to maintain without close collaboration with SRI. The choice of Isabelle as an alternative platform may seem obvious with its extensive support for the various aspects of encoding an object logic. Still, it is important not to overlook the requirement for decision procedure support for real arithmetic. At the time this support was one of the primary reasons for choosing PVS as a platform for PC/DC and the requirement also applies for a tool based on Isabelle. Only the integration with a decision procedure like SVC makes Isabelle a practicable choice as a DC proof tool platform.

10.2.3 Duration Calculus Encoding

The encoding is the basis of the tool. It must be sound and faithful to the original formulation but also practical for proof work. Our work on the Steam Boiler demonstrates that the encoding satisfies these requirements. Still, we can look for improvements in the way we use Isabelle and comment on the similarity of the encoding to the one found in PC/DC.

Formula Layers

Basic DC has two layers of formulas, logical and state, and speak only of boolean states. Our encoding started with this limitation representing logical formulas as type `o` and state formulas as type `so`. Meta-predicates `RT` and `CF` was (and is) used to encode syntactical side-conditions of rigid terms and chop-free formulas. This worked well, but the extension to typed states necessitated the introduction of a third layer of rigid formulas. The reason was the advent of global variables in state formulas under the duration sign and the need to avoid substitution of these global variables for flexible terms. Since we do substitution in our object logic by meta logical β-conversion we need to encode the distinction between rigid and modal terms in the type system and introduced the type `ro`. The meta-predicate `RT` is no longer strong enough for this purpose.

Concluding that we need the third level it leaves the question if the predicate `RT` is really needed. We have ended up having two different ways to characterize rigid terms. The rules of the `RT` predicate is one, the rewriting system surrounding the lifting functions (`nr, no`) is the other, given that a real term is rigid if it can be rewritten in to `nr(`x`)` for some x.

We definitely need the lifting functions as they link the levels of rigid and modal terms, but the future of the `RT` might be re-considered.

The `RT` does add flexibility, however. Using only the rewriting system of the lifting functions requires that the system is suitably powerful. Functions and operators symbols must have representations at both levels. This is the case for the arithmetic operators for example, but should also be the case for any user introduced functions and operators otherwise rewriting into the $n(x)$ form will not generally be possible. The `RT` proof system only requires proof rules for functions and operators at the modal term level, not declaration of these functions at both levels and accompanying rewriting rules.

Quantification Over Modal Variables

We allow quantification over modal variables. While this allow for proofs of simple chop-free lemmas like $\vdash (\exists x \cdot \ell = x)$ it also necessitates controlling instantiation with suitable side-conditions on the quantifier rules. Having introduced the class of rigid terms we can consider changing the declaration of the quantifiers to allow only quantification over types in this class. This will allow us to simplify the quantifier rules by dropping the side conditions. Doing so will have consequences for the power of the proof system which must be investigated. Clearly $\vdash (\exists x \cdot \ell = x)$ or generally $\vdash (\exists x \cdot \int P = x)$ must be added as an axiom.

Constructing Modal and State Terms

In the present encoding we have not exploited the features of the Isabelle type system to its fullest. Having settled on a three layer term structure we might as well seek to reflect their relation in the Isabelle type system. This is possible through the use of type constructors. Instead of having three unrelated types `ro`, `o` and `so`, we can construct `o` and `so` from `ro` using type constructors `modal` and `state`:

```
types
  'a modal
  'a state

arities
  modal :: (rterm)fterm
  state :: (rterm)sterm
```

```
types
  o = ro modal
  so = ro state
```

Such an approach will allow us to generalize the lifting functions to have only one (n) for lifting to modal terms and one (s) for lifting to state terms. This again we save us the need to restate the induction rules for each rigid type.

Relation to PVS Encoding

We have characterized the Isabelle encoding as syntactic and the PVS encoding as semantic. Such a conventional characterization seems to indicate a greater difference in the encoding than is actually present. Basic DC only needs two term levels in the Isabelle formulation but with the introduction of typed states both encodings are three level with rigid, modal and state terms.

Modal and state terms are concrete functions spaces in PC/DC, making it a true semantic encoding. An advantage is that we can prove soundness of rules and axioms within the system itself. Other than that, there is a one-to-one relationship between declared constants and types in the two encodings. They both suffer from the nuisance of lifting functions but at least PVS improves somewhat on this by featuring automatic conversions. PVS also has a flexible subtype concept.

One advantage of the Isabelle/DC encoding is that reasoning is carried out at the modal formula level. No complications arise from having to hack the proof commands of PVS to work on DC formulas by clever rewriting and selective unparsing.

Non of the encodings are useful for meta-theoretical reasoning about the proof system itself. They both lack an encoding of what a proof is, and it is not possible to induct over formulas and terms so frequently needed in this kind of work.

10.3 Proof Support

We had the aim of improving the proof support provided by Isabelle/DC over PC/DC and we have focussed on the aspect of reducing proof size. To be able to judge if any improvement was achieved we chose the Steam Boiler verification as a benchmark for our work. By doing this verification in both tools we get the basis for a quantitative comparison. The reduction factor of two in the size of Isabelle/DC proofs substantiates the subjective impression from using Isabelle/DC: Proof development is faster and proof maintenance is easier. We also conclude from the obtained data that to further reduce the proof size we must look for tactics doing extended proof search.

10.4 Future Work

A number of subjects for future work was discussed above and throughout the thesis. Here we summarize them in a bulleted list:

- Introduce the tactics and solvers used in the Steam Boiler case study back in to Isabelle/DC so that new applications can benefit from them.
- Change the encoding of modal terms and state terms by providing these types via type constructors from their corresponding rigid types. This would allow a generalization of lifting functions such as `nr` and `sr` which in turn would allow the induction rules to be expressible for any rigid domain.
- Implement a convenient concrete syntax for real literals.
- Implement basic support for treatment of non-linear real arithmetic. We are here thinking of a large collection of lemmas resembling the collection found in the PVS prelude.
- Develop a more comprehensive theory for the EDC operators and their relation to the duration operator.
- Implement decision procedure for DC subsets. NFDC is one such subset for which a prototype implementation already exists. This is based on the real time automata tool Uppaal [30]. A larger, but still decidable, subset is the subset of requirements and design formulas investigated by Lakhnech [28] a subset containing *DC Implementables* as investigated by Ravn.
- With the Steam Boiler proofs as a starting point look for ways of automatic controlling proof steps by employing proof search.
- Extend Isabelle/DC with a collection of datatypes or facilities for defining such. Inspiration for this work can be found in the Isabelle theory HOL.

Appendix A

Isabelle/DC Source Code

This appendix contains listings of the Isabelle theories forming the encoding of Duration Calculus presented in the thesis. We also present the ML code implementing the interface to the oracles used. The soundness of the tool relies on this encoding and the correctness of the interfaces (and the decision procedures of course, but they are of out of scope of this presentation). Proofs scripts of derived rules and definitions of tactic and tacticals are left out for lack of space.

A.1 Sequence

```
Sequence = Pure +

classes
  term < logic
  fterm < term

types
  o

arities
  o :: fterm

(* Sequences *)

types
 seq'

consts
 Seq0'         :: "[o,seq']=>seq'"
 Seq1'         :: "o=>seq'"


(* concrete syntax *)
```

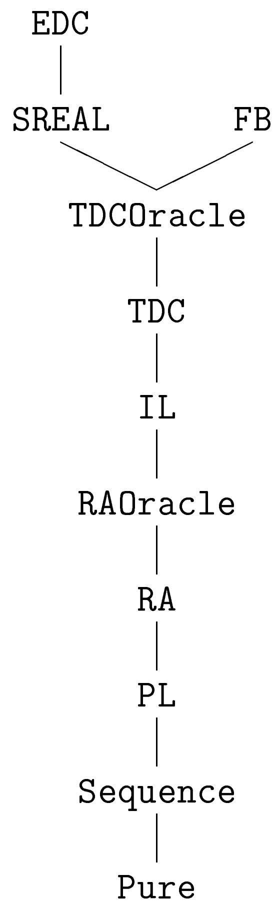


Figure A.1: Hierarchy of Isabelle/DC theories

```
types
  seq seqobj seqcont


syntax
 SeqEmp         :: "seq"                           ("")
 SeqApp         :: "[seqobj,seqcont] => seq"       ("__")

 SeqContEmp     :: "seqcont"                       ("")
 SeqContApp     :: "[seqobj,seqcont] => seqcont"   (",/ __")

 SeqO           :: "o => seqobj"                   ("_")
 SeqId          :: "id => seqobj"                  ("$_")
 SeqVar         :: "var => seqobj"                 ("$_")

types

 single_seqe = "[seq,seqobj] => prop"
 single_seqi = "[seq'=>seq',seq'=>seq'] => prop"
 two_seqi = "[seq'=>seq', seq'=>seq'] => prop"
 two_seqe = "[seq, seq] => prop"
 three_seqi = "[seq'=>seq', seq'=>seq', seq'=>seq'] => prop"
 three_seqe = "[seq, seq, seq] => prop"
 four_seqi = "[seq'=>seq', seq'=>seq', seq'=>seq', seq'=>seq'] => prop"
 four_seqe = "[seq, seq, seq, seq] => prop"

end

ML

(* parse translation for sequences *)

fun abs_seq' t = Abs("s", Type("seq'",[]), t);

fun seqobj_tr(Const("SeqO",_)$f) = Const("SeqO'",dummyT)$f |
    seqobj_tr(_$i) = i;

fun seqcont_tr(Const("SeqContEmp",_)) = Bound 0 |
    seqcont_tr(Const("SeqContApp",_)$so$sc) =
      (seqobj_tr so)$(seqcont_tr sc);

fun seq_tr(Const("SeqEmp",_)) = abs_seq'(Bound 0) |
    seq_tr(Const("SeqApp",_)$so$sc) =
      abs_seq'(seqobj_tr(so)$seqcont_tr(sc));


fun singlobj_tr(Const("SeqO",_)$f) =
    abs_seq' ((Const("SeqO'",dummyT)$f)$Bound 0);
```

```
(* print translation for sequences *)

fun seqcont_tr' (Bound 0) =
      Const("SeqContEmp",dummyT) |
    seqcont_tr' (Const("SeqO'",_)$f$s) =
      Const("SeqContApp",dummyT)$
      (Const("SeqO",dummyT)$f)$
      (seqcont_tr' s) |
(*    seqcont_tr' ((a as Abs(_,_,_))$s)=
      seqcont_tr'(betapply(a,s)) | *)
    seqcont_tr' (i$s) =
      Const("SeqContApp",dummyT)$
      (Const("SeqId",dummyT)$i)$
      (seqcont_tr' s);

fun seq_tr' s =
    let fun seq_itr' (Bound 0) =
              Const("SeqEmp",dummyT) |
            seq_itr' (Const("SeqO'",_)$f$s) =
              Const("SeqApp",dummyT)$
              (Const("SeqO",dummyT)$f)$(seqcont_tr' s) |
(*            seq_itr' ((a as Abs(_,_,_))$s) =
              seq_itr'(betapply(a,s)) |    *)
            seq_itr' (i$s) =
              Const("SeqApp",dummyT)$
              (Const("SeqId",dummyT)$i)$
              (seqcont_tr' s)
    in case s of
         Abs(_,_,t) => seq_itr' t |
         _ => s$(Bound 0)
    end;




fun single_tr c [s1,s2] =
    Const(c,dummyT)$seq_tr s1$singlobj_tr s2;

fun two_seq_tr c [s1,s2] =
    Const(c,dummyT)$seq_tr s1$seq_tr s2;

fun three_seq_tr c [s1,s2,s3] =
    Const(c,dummyT)$seq_tr s1$seq_tr s2$seq_tr s3;

fun four_seq_tr c [s1,s2,s3,s4] =
    Const(c,dummyT)$seq_tr s1$seq_tr s2$seq_tr s3$seq_tr s4;


fun singlobj_tr'(Const("SeqO'",_)$fm) = fm |
    singlobj_tr'(id) = Const("@SeqId",dummyT)$id;
```

```
fun single_tr' c [s1, s2] =
(Const (c, dummyT)$seq_tr' s1$seq_tr' s2 );


fun two_seq_tr' c [s1, s2] =
  Const (c, dummyT)$seq_tr' s1$seq_tr' s2;

fun three_seq_tr' c [s1, s2, s3] =
  Const (c, dummyT)$seq_tr' s1$seq_tr' s2$seq_tr' s3;

fun four_seq_tr' c [s1, s2, s3, s4] =
  Const (c, dummyT)$seq_tr' s1$seq_tr' s2$seq_tr' s3$seq_tr' s4;
```

A.2 PL

```
PL = Sequence +

classes
  rterm < term  (* rigid terms *)
  prp (* type with propositional operators *)

default
  term

types
  ro  (* rigid formulas *)

arities
  ro :: rterm

  o :: prp
  ro :: prp

consts

  (* Coercion to prop *)
  Trueprop      :: "two_seqi"
  "@Trueprop"   :: "two_seqe" ("((_)/ |- (_))" [6,6] 5)

  (* Propositional constants and operators *)
  True,False   :: 'a::prp
  Not          :: 'a::prp => 'a            ("~ _" [40] 40)
  "&"          :: ['a::prp,'a] => 'a       (infixr 35)
  "|"          :: ['a::prp,'a] => 'a       (infixr 30)
  "-->","<->"  :: ['a::prp,'a] => 'a       (infixr 25)
```

```
  no          :: ro => o

  (* Quantifiers *)
  All         :: ('a => o) => o           (binder "ALL " 10)
  Ex          :: ('a => o) => o           (binder "EX " 10)

  (* Equality *)
  "="         :: ['a::fterm,'a] => o      (infixl 50)
  ":="        :: ['a::rterm,'a] => ro     (infixl 50)

  (* Side condition predicates *)
  CF          :: o => prop         ("CF _")  (* Chop Free *)
  RT          :: 'a => prop        ("RT _") (* Rigid Term *)

rules

  (*Structural rules*)

  contRS "$H |- $E, $S, $S, $F ==> $H |- $E, $S, $F"
  contLS "$H, $S, $S, $G |- $E ==> $H, $S, $G |- $E"

  thinRS "$H |- $E, $F ==> $H |- $E, $S, $F"
  thinLS "$H, $G |- $E ==> $H, $S, $G |- $E"

  exchRS "$H |- $E, $R, $S, $F ==> $H |- $E, $S, $R, $F"
  exchLS "$H, $R, $S, $G |- $E ==> $H, $S, $R, $G |- $E"

  (*Propositional rules*)

  basic  "$H, A, $G |- $E, A, $F"

  conjR "[| $H|- $E, P, $F;  $H|- $E, Q, $F |] ==> $H|- $E, P&Q, $F"
  conjL "$H, P, Q, $G |- $E ==> $H, P & Q, $G |- $E"

  disjR "$H |- $E, P, Q, $F ==> $H |- $E, P|Q, $F"
  disjL "[| $H, P, $G |- $E;  $H, Q, $G |- $E |] ==> $H, P|Q, $G |- $E"

  impR  "$H, P |- $E, Q, $F ==> $H |- $E, P-->Q, $F"
  impL  "[| $H,$G |- $E,P;  $H, Q, $G |- $E |] ==> $H, P-->Q, $G |- $E"

  notR  "$H, P |- $E, $F ==> $H |- $E, ~P, $F"
  notL  "$H, $G |- $E, P ==> $H, ~P, $G |- $E"

  FalseL "$H, False, $G |- $E"

  cut   "[| $H |- $E, P;  $H, P |- $E |] ==> $H |- $E"

  True_def "True == False-->False"
  iff_def  "P<->Q == (P-->Q) & (Q-->P)"

  (* Quantifiers *)
```

```
allR  "(!!x. RT x ==> $H |- $E, P(x), $F) ==> $H |- $E, ALL x. P(x), $F"
allLRT "[| $H, P(x), $G, ALL x. P(x) |- $E; RT x |] ==>
             $H, ALL x. P(x), $G |- $E"
allLCF "[| $H, P(x), $G, ALL x. P(x) |- $E; CF(P(x)) |] ==>
             $H, ALL x. P(x), $G |- $E"

exRRT "[| $H |- $E, P(x), $F, EX x. P(x); RT x |] ==>
             $H |- $E, EX x. P(x), $F"
exRCF "[| $H |- $E, P(x), $F, EX x. P(x); CF(P(x)) |] ==>
             $H |- $E, EX x. P(x), $F"
exL   "(!!x. RT x ==> $H, P(x), $G |- $E) ==> $H, EX x. P(x), $G |- $E"


(*Equality*)

refl  "$H |- $E, a=a, $F"

repl  "CF A(x) ==> $I, a=b, $H, A(a), $G |- $E, A(b), $F"

eq_iff "A=B == A<->B"
req_iff "A:=B == A<->B"

(* Reflection *)

eq_reflection  "|- x=y ==> (x==y)"
iff_reflection "|- A<->B ==> (A==B)"

(* Chop Free *)

CFno     "CF(no(x))"
CFtrue   "CF(True)"
CFfalse  "CF(False)"
CFnot    "CF(A) ==> CF(~ A)"
CFand    "[| CF(A); CF(B) |] ==> CF(A&B)"
CFor     "[| CF(A); CF(B) |] ==> CF(A|B)"
CFimp    "[| CF(A); CF(B) |] ==> CF(A-->B)"
CFiff    "[| CF(A); CF(B) |] ==> CF(A<->B)"

CFeq     "CF(x = y)"

CFall    "CF(A(x)) ==> CF(ALL x. A(x))"
CFex     "CF(A(x)) ==> CF(EX x. A(x))"

(* Rigid Formula *)

notrue   "True == no(True)"
nofalse  "False == no(False)"
nonot    "~no(A) == no(~ A)"
noand    "no(A) & no(B) == no(A & B)"
noor     "no(A) | no(B) == no(A | B)"
```

```
  noimp    "no(A) --> no(B) == no(A --> B)"
  noiff    "no(A) <-> no(B) == no(A <-> B)"

  noeq_o   "no(A) = no(B) == no(A := B)"

  noexists "RT x ==> |- (EX y . x=no(y))"

  RTrterm  "RT(x::'a::rterm)"

  RTno     "RT(no(x))"

  RTtrue   "RT(True::o)"
  RTfalse  "RT(False::o)"
  RTnot    "RT(A::o) ==> RT(~ A)"
  RTand    "[| RT(A::o); RT(B) |] ==> RT(A&B)"
  RTor     "[| RT(A::o); RT(B) |] ==> RT(A|B)"
  RTimp    "[| RT(A::o); RT(B) |] ==> RT(A-->B)"
  RTiff    "[| RT(A::o); RT(B) |] ==> RT(A<->B)"

  RTall    "(RT x ==> RT(A(x))) ==> RT(ALL x. A(x))"
  RTex     "(RT x ==> RT(A(x))) ==> RT(EX x. A(x))"

  RTeq     "[| RT x; RT y |] ==> RT(x=y)"

end

ML

val parse_translation = [("@Trueprop",Sequence.two_seq_tr "Trueprop")];
val print_translation = [("Trueprop",Sequence.two_seq_tr' "@Trueprop")];

val thy_data = [Simplifier.simpset_thy_data];
```

A.3 RA

```
RA= PL +

classes
  arith

default
  arith

types
  real
  rreal

arities
  real :: arith
  rreal :: arith
```

```
  real :: fterm
  rreal :: rterm

consts
  (* real terms *)

  zero          :: 'a                          ("0")
  one           :: 'a                          ("1")

  "+"           :: ['a, 'a] => 'a              (infixl 60)
  "-"           :: ['a, 'a] => 'a              (infixl 60)
  "--"          :: 'a => 'a                    ("-- _" [80] 80)
  "*"           :: ['a, 'a] => 'a              (infixl 70)
  "op /"        :: ['a, 'a] => 'a              ("(_ '// _)" [70,71] 70)

  "<"           :: [real, real] => o           (infixr 50)
  "<="          :: [real, real] => o           (infixr 50)
  ">"           :: [real, real] => o           (infixr 50)
  ">="          :: [real, real] => o           (infixr 50)

  ":<"          :: [rreal, rreal] => ro        (infixr 50)
  ":<="         :: [rreal, rreal] => ro        (infixr 50)
  ":>"          :: [rreal, rreal] => ro        (infixr 50)
  ":>="         :: [rreal, rreal] => ro        (infixr 50)

  nr            :: rreal => real

defs
  le_def        "x<=y == x<y | x=y"
  gt_def        "x>y == y<x"
  ge_def        "x>=y == y<=x"
  rle_def       "x:<=y == x:<y | x:=y"
  rgt_def       "x:>y == y:<x"
  rge_def       "x:>=y == y:<=x"

syntax
  two           :: 'a                          ("2")
  three         :: 'a                          ("3")
  four          :: 'a                          ("4")
  five          :: 'a                          ("5")
  six           :: 'a                          ("6")
  seven         :: 'a                          ("7")
  eight         :: 'a                          ("8")
  nine          :: 'a                          ("9")

translations
  "2" == "1+1"
  "3" == "2+1"
  "4" == "3+1"
  "5" == "4+1"
  "6" == "5+1"
```

```
  "7" == "6+1"
  "8" == "7+1"
  "9" == "8+1"

rules
  (* This formulation is taken from the prelude of PVS *)

  (* Literals *)

  zero_ne_one          "|- ~ (0 = 1)"
  zero_lt_one          "|- 0 < 1"

  (* Field Axioms *)

  commutative_add      "|- x + y = y + x"

  associative_add      "|- x + (y + z) = (x + y) + z"

  identity_add         "|- x + 0 = x"

  inverse_add          "|- x + --x = 0"

  minus_add            "|- x - y = x + --y"

  commutative_mult     "|- x * y = y * x"

  associative_mult     "|- x * (y * z) =  (x * y) * z"

  identity_mult        "|- 1 * x = x"

  inverse_mult         "|- ~(x=0) --> x * (1 / x) = 1"

  div_def              "|- ~(x=0) --> y / x = y * (1 / x)"

  distributive         "|- x * (y + z) = (x * y) + (x * z)"

  (* Order Axioms *)

  posreal_add_closed   "|- x > 0 & y > 0 --> x + y > 0"

  posreal_mult_closed  "|- x > 0 & y > 0 --> x * y > 0"

  posreal_neg          "|- x > 0 --> ~(--x > 0)"

  trichotomy           "|- x > 0 | x = 0 | 0 > x"


  (* Rigid terms *)

  nrzero        "0 == nr(0)"
  nrone         "1 == nr(1)"
```

```
  nradd         "nr(x) + nr(y) == nr(x + y)"
  nrsub         "nr(x) - nr(y) == nr(x - y)"
  nrneg         "--(nr(x)) == nr(--x)"
  nrmult        "nr(x) * nr(y) == nr(x * y)"
  nrdiv         "nr(x) / nr(y) == nr(x / y)"

  nolt          "nr(x) < nr(y) == no(x :< y)"
  nole          "nr(x) <= nr(y) == no(x :<= y)"
  nogt          "nr(x) > nr(y) == no(x :> y)"
  noge          "nr(x) >= nr(y) == no(x :>= y)"
  noeq_real     "nr(x) = nr(y) == no(x := y)"

  nrexists      "RT x ==> |- (EX y . x=nr(y))"

  RTnr          "RT(nr(x))"

  RTzero        "RT(0)"
  RTone         "RT(1)"

  RTadd         "[| RT(x); RT(y) |] ==> RT(x+y)"
  RTsub         "[| RT(x); RT(y) |] ==> RT(x-y)"
  RTmult        "[| RT(x); RT(y) |] ==> RT(x*y)"
  RTdiv         "[| RT(x); RT(y) |] ==> RT(x / y)"
  RTneg         "RT x ==> RT(--x)"

  RTlt          "[| RT(x); RT(y) |] ==> RT(x<y)"
  RTle          "[| RT(x); RT(y) |] ==> RT(x<=y)"
  RTgt          "[| RT(x); RT(y) |] ==> RT(x>y)"
  RTge          "[| RT(x); RT(y) |] ==> RT(x>=y)"

end
```

A.4 RAOracle

```
RAOracle = "declRAOracle" + RA +
oracle RA_oracle = RA_oracle
end
```

A.4.1 RA Oracle Implementation

```
use "svc.ML";

(*New exception constructor for passing arguments to the oracle*)

exception RAOracleExn of term;

fun Expr_of t =
    let
```

```
        open Svc
        val len_expr = uninterp_expr("len", [])
        val zero_expr = int_expr 0
        fun i_of (Const("op +", T) $ A $ B) = (i_of A) + (i_of B)
          | i_of (Const("op -", T) $ A $ B) = (i_of A) - (i_of B)
          | i_of (Const("op *", T) $ A $ B) = (i_of A) * (i_of B)
          | i_of (Const("--", _) $ A) = ~(i_of A)
          | i_of (Const("zero", _)) = 0
          | i_of (Const("one", _)) = 1
          | i_of (Const("nr", _) $ A) = i_of A
        fun t_of (Const("op +", T) $ A $ B) = interp_expr("+", [t_of A, t_of B])
          | t_of (Const("op -", _) $ A $ B) =
            interp_expr("+", [t_of A, interp_expr("*", [int_expr ~1, t_of B])])
          | t_of (Const("op *", _) $ A $ B) = interp_expr("*", [t_of A, t_of B])
          | t_of (Const("op div", _) $ A $ B) =
            interp_expr("*", [t_of A, rat_expr(1, i_of B)])
          | t_of (Const("--", _) $ A) = interp_expr("*", [int_expr ~1, t_of A])
          | t_of (Const("zero", _)) = int_expr 0
          | t_of (Const("one", _)) = int_expr 1
          | t_of (Const("nr", _)  $ A) = t_of A
          | t_of (Var((id, 0), _)) = uninterp_expr(id, [])
        fun f_of (Const("Trueprop", _) $ A) = f_of A
          | f_of (Const("op &", _) $ A $ B) = buildin_expr("AND", [f_of A, f_of B])
          | f_of (Const("op |", _) $ A $ B) = buildin_expr("OR", [f_of A, f_of B])
          | f_of (Const("op <->", _) $ A $ B) = buildin_expr("<=>", [f_of A, f_of B])
          | f_of (Const("op -->", _) $ A $ B) = buildin_expr("=>", [f_of A, f_of B])
          | f_of (Const("Not", _) $ A) = buildin_expr("NOT", [f_of A])
          | f_of (Const("True", _)) = true_expr
          | f_of (Const("False", _)) = false_expr
          | f_of (Const("op =", _) $ x $ y) = buildin_expr("=", [t_of x, t_of y])
          | f_of (Const("op <", _) $ x $ y) = buildin_expr("<", [t_of x, t_of y])
          | f_of (Const("op <=", _) $ x $ y) = buildin_expr("<=", [t_of x, t_of y])
          | f_of (Const("op >", _) $ x $ y) = buildin_expr(">", [t_of x, t_of y])
          | f_of (Const("op >=", _) $ x $ y) = buildin_expr(">=", [t_of x, t_of y])
          | f_of (Const("op :=", _) $ x $ y) = buildin_expr("=", [t_of x, t_of y])
          | f_of (Const("op :<", _) $ x $ y) = buildin_expr("<", [t_of x, t_of y])
          | f_of (Const("op :<=", _) $ x $ y) = buildin_expr("<=", [t_of x, t_of y])
          | f_of (Const("op :>", _) $ x $ y) = buildin_expr(">", [t_of x, t_of y])
          | f_of (Const("op :>=", _) $ x $ y) = buildin_expr(">=", [t_of x, t_of y])
          | f_of (Const("no", _)  $ A) = f_of A
          | f_of (Var((id, 0), _)) = uninterp_expr(id, [])
    in
        f_of t
    end;


fun RA_oracle (sign, RAOracleExn svc_form) =
    if Svc.valid (Expr_of svc_form) then
        c_Trueprop $ (sequence []) $ (sequence [svc_form])
    else
        raise RAOracleExn svc_form;
```

A.4.2 SVC Interface

```
structure Svc =
struct
    datatype Expr =
        bracketed_expr of Expr
      | ref_def_expr of string * Expr
      | ref_expr of string
      | typed_expr of Type * Expr
      | buildin_expr of string * Expr list
      | interp_expr of string * Expr list
      | uninterp_expr of string * Expr list
      | false_expr
      | true_expr
      | distinct_expr of string
      | int_expr of int
      | rat_expr of int * int
    and Type =
        simple_type of string
      | array_type of Type * Type
      | record_type of (Expr * Type) list
      | bitvec_type of int


    fun tostring t =
        let fun intToString i =
            if i < 0 then
                "-"^(Int.toString (~i))
            else
                (Int.toString i)
            fun ut(simple_type s) = s ^ " "
              | ut(array_type(t1, t2)) = "ARRAY " ^ (ut t1) ^ (ut t2)
              | ut(record_type fl) =
                "RECORD" ^
                (foldl (fn (a, (d, t)) => a ^ (ue d) ^ (ut t)) (" ", fl))
              | ut(bitvec_type n) = "BITVEC " ^ (Int.toString n) ^ " "
            and ue(bracketed_expr e) = "(" ^ (ue e) ^ ") "
              | ue(ref_def_expr(r, e)) = "$" ^ r ^ ":" ^ (ue e)
              | ue(ref_expr r) = "$" ^ r ^ " "
              | ue(typed_expr(t, e)) = (ut t) ^ (ue e)
              | ue(buildin_expr(s, l)) =
                "(" ^ s ^ (foldl (fn (a, b) => a ^ " " ^ (ue b)) ("", l)) ^ ") "
              | ue(interp_expr(s, l)) =
                "{" ^ s ^ (foldl (fn (a, b) => a ^ " " ^ (ue b)) ("", l)) ^ "} "
              | ue(uninterp_expr(s, l)) =
                "(" ^ s ^ (foldl (fn (a, b) => a ^ " " ^ (ue b)) ("", l)) ^ ") "
              | ue(false_expr) = "FALSE "
              | ue(true_expr) = "TRUE "
```

```
              | ue(distinct_expr s) = "@" ^ s ^ " "
              | ue(int_expr i) = (intToString i) ^ " "
              | ue(rat_expr(i, j)) = (intToString i) ^ "|" ^ (intToString j) ^ " "
        in
            ue t
        end

  fun valid e =
      let fun getenv var default =
          case OS.Process.getEnv var of
              SOME(s) => s
            | NONE => (print (var^" not set! Using "^default^"\n");
                         default)
          val home = getenv "HOME" "/home/sh"
          val svc_home = getenv "SVC_HOME" (home^"/svc-1.0")
          val svc_machine = getenv "SVC_MACHINE" "i386-redhat-linux"
          val check_valid = svc_home^"/"^svc_machine^"/bin/check_valid"
          val svc_input = tostring e
          val svc_input_file = OS.FileSys.tmpName()
          val svc_input_strm = TextIO.openOut svc_input_file
          val svc_output_file = OS.FileSys.tmpName()

          val _ = TextIO.outputSubstr(svc_input_strm, Substring.all svc_input)
          val _ = TextIO.closeOut svc_input_strm
          val _ =
              OS.Process.system
              (check_valid^" -dump-result "^svc_output_file^
               " "^svc_input_file^"> /dev/null 2>&1")
          val svc_output_strm = TextIO.openIn svc_output_file
          val svc_output = TextIO.inputLine svc_output_strm
          val _ = TextIO.closeIn svc_output_strm
      in
          String.isPrefix "VALID" svc_output
      end
end;
```

A.5 IL

```
IL = RAOracle +

consts

  (* Modalities *)

  "^"           :: [o, o] => o                    (infixr 45)
  dia           ::  o => o                        ("<>_" [50] 50)
  box           ::  o => o                        ("[]_" [50] 50)
  len           :: real

  "@Lstar"      :: "two_seqe"                     ("(_)|L>(_)" [6,6] 5)
```

```
  "@Rstar"      :: "two_seqe"                    ("(_)|R>(_)" [6,6] 5)
  Lstar,Rstar   :: "two_seqi"

rules

  (* Definitions *)

  dia_def       "<>A == True^A^True"
  box_def       "[]A == ~(<>(~A))"

  (* IL axioms [Zhou and Hansen] *)

  A0            "|- len >= 0"

  A1a           "|- A^B & ~(A^C) --> A^(B & ~C)"
  A1b           "|- A^B & ~(C^B) --> (A & ~C)^B"

  A2            "|- (A^B)^C <-> A^B^C"

  Ra            "RT(A) ==> |- A^B --> A"
  Rb            "RT(B) ==> |- A^B --> B"

  Ba            "|- (EX x. A(x))^B --> (EX x. A(x)^B)"
  Bb            "|- A^(EX x. B(x)) --> (EX x. A^B(x))"

  L1a           "RT x ==> |- (len=x)^A --> ~((len=x)^(~A))"
  L1b           "RT x ==> |- A^(len=x) --> ~((~A)^(len=x))"

  L2            "[| RT x; RT y |] ==>
                  |- x >= 0 & y >= 0 --> ((len=x+y) <-> (len=x)^(len=y))"

  L3            "|- A --> A^(len=0)"

  L4            "|- A --> (len=0)^A"

  (* IL rules *)
  (* Prefixing a variable with $ makes it range over sequences of formulas *)

  monotone_chop "$G |- $E, [](A-->B), $F  ==>  $G |- $E, A^C-->B^C, $F"

  chop_monotone "$G |- $E, [](A-->B), $F  ==>  $G |- $E, C^A-->C^B, $F"

  boxR          "[| $G |L> $G'; $E |R> $E'; $F |R> $F';
                       $G' |- $E', P, $F'|]  ==>  $G |- $E, []P, $F"

  (* Rigid formulas and terms *)

  RTchop        "[| RT(A); RT(B) |] ==> RT(A^B)"
  RTbox         "RT(A) ==> RT([]A)"
  RTdia         "RT(A) ==> RT(<>A)"
```

```
  (* Definition of the star operation using a set of Horn clauses *)
  (* For system S4:  gamma * == {[]P | []P : gamma}              *)
  (*                 delta * == {<>P | <>P : delta}              *)

  lstar0         "|L>"
  lstar1         "$G |L> $H ==> []P, $G |L> []P, $H"
  lstar2         "$G |L> $H ==>   P, $G |L>      $H"
  lstar3         "[| $G |L> $H; $E |L> $F |] ==>  $E, $G |L>  $F, $H"
  lstarRT        "[| $G |L> $H; RT P |] ==>  P, $G |L> P, $H"

  rstar0         "|R>"
  rstar1         "$G |R> $H ==> <>P, $G |R> <>P, $H"
  rstar2         "$G |R> $H ==>   P, $G |R>      $H"
  rstar3         "[| $G |R> $H; $E |R> $F |] ==>  $E, $G |R>  $F, $H"
  rstarRT        "[| $G |R> $H; RT P |] ==>  P, $G |R> P, $H"

  lstar4         "$G |L> $H ==>  $S, $G |L>      $H"
  rstar4         "$G |R> $H ==>  $S, $G |R>      $H"

end

ML

local

  val Lstar = "Lstar";
  val Rstar = "Rstar";
  val SLstar = "@Lstar";
  val SRstar = "@Rstar";

  fun star_tr c [s1,s2] =
      Const(c,dummyT)$Sequence.seq_tr s1$Sequence.seq_tr s2;
  fun star_tr' c [s1,s2] =
      Const(c,dummyT) $ Sequence.seq_tr' s1 $ Sequence.seq_tr' s2;
in
val parse_translation = [(SLstar,star_tr Lstar), (SRstar,star_tr Rstar)];
val print_translation = [(Lstar,star_tr' SLstar), (Rstar,star_tr' SRstar)]
end;
```

A.6 TDC

```
TDC = IL +

classes
  sterm < logic

default
  sterm
```

```
types
  so

arities
  so :: sterm

consts

  (* Duration *)

  dur           :: so => real
  pr            :: so => o                         ("‘_‘")
  point         :: o                               ("‘‘")

  (* State predicates *)

  tt            :: so                              ("tt")
  ff            :: so                              ("ff")

  (* Connectives *)

  NOT           :: so => so                        ("NOT _" [40] 40)
  AND           :: [so, so] => so                  (infixr 35)
  OR            :: [so, so] => so                  (infixr 30)
  IMPLIES       :: [so, so] => so                  (infixr 25)
  IFF           :: [so, so] => so                  (infixr 25)

  (* equaility *)

  ";="         :: ['a, 'a] => so                   (infixl 50)

  (* Lifting *)

  Oo             :: so => o                        ("Oo _")

  so             :: ro => so

rules

  (* Definitions *)

  len_def        "len == dur(tt)"
  pr_def         "‘s‘ == dur(s) = len & len > 0"
  point_def      "‘‘ == len = 0"

  (* Definitions *)

  stt_def       "tt      == ff IMPLIES ff"
  snot_def      "NOT P   == P IMPLIES ff"
  siff_def      "P IFF Q == (P IMPLIES Q) AND (Q IMPLIES P)"
```

```
  (* DC *)

  DCA1          "|- dur(ff) = 0"

  DCA2          "|- dur(tt) = len"

  DCA3          "|- dur(A) >= 0"

  DCA4          "|- dur(A) + dur(B) = dur(A OR B) + dur(A AND B)"

  DCA5          "[|RT x; RT y|] ==> |- (dur(A)=x) ^ (dur(A)=y) --> dur(A)=x+y"

  DCA6          "$G' |- $E', []((Oo A)<->(Oo B)), $F'
                 ==>  $G |- $E, dur(A)=dur(B), $F"

  induct_chop   "[| $G |- H(``), $D;
                    !! X . $G |- H(X) --> H(X|(`A`^X)|(`NOT A`^X)), $D
                 |] ==> $G |- H(True), $D"
  chop_induct   "[| $G |- H(``), $D;
                    !! X . $G |- H(X) --> H(X|(X^`A`)|(X^`NOT A`)), $D
                 |] ==> $G |- H(True), $D"

  (* Reflection *)

  P_reflection  "|- `A` <-> `B` ==> (A == B)"

  (* Lifting *)

  Ooeq_o         "Oo((x::so) ;= y) == (Oo x) = (Oo y)"

  Ooso           "Oo(so(x)) == no(x)"

  Oott           "Oo tt == True"
  Ooff           "Oo ff == False"

  Oonot          "Oo(NOT A) == ~(Oo A)"
  Ooand          "Oo(A AND B) == (Oo A) & (Oo B)"
  Ooor           "Oo(A OR B) == (Oo A) | (Oo B)"
  Ooimplies      "Oo(A IMPLIES B) == (Oo A) --> (Oo B)"
  Ooiff          "Oo(A IFF B) == (Oo A) <-> (Oo B)"

  (* Chop free *)

  CFpr           "CF `A`"
  CFpoint        "CF ``"

end
```

A.7 TDCOracle

```
TDCOracle = "declTDCOracle" + TDC +
oracle TDC_oracle = TDC_oracle
end
```

A.7.1 DC Oracle Implementation

```
use "isachecking.sml";

(*New exception constructor for passing arguments to the oracle*)
exception TDCOracleExn of Formula.Formula;

fun Formula_of t =
    let
        open Formula
        fun s_of (Const("op AND", _) $ A $ B) = sAnd(s_of A, s_of B)
          | s_of (Const("op OR", _) $ A $ B) = sOr(s_of A, s_of B)
          | s_of (Const("op IFF", _) $ A $ B) = sIff(s_of A, s_of B)
          | s_of (Const("op IMPLIES", _) $ A $ B) = sImp(s_of A, s_of B)
          | s_of (Const("NOT", _) $ A) = sNeg(s_of A)
          | s_of (Const("tt", _)) = tt
          | s_of (Const("ff", _)) = ff
          | s_of (Var((id, 0), _)) = Id(id)
        fun f_of (Const("Trueprop", _) $ A) = f_of A
          | f_of (Const("op &", _) $ A $ B) = And(f_of A, f_of B)
          | f_of (Const("op |", _) $ A $ B) = Or(f_of A, f_of B)
          | f_of (Const("op <->", _) $ A $ B) = Iff(f_of A, f_of B)
          | f_of (Const("op -->", _) $ A $ B) = Imp(f_of A, f_of B)
          | f_of (Const("Not", _) $ A) = Neg(f_of A)
          | f_of (Const("op ^", _) $ A $ B) = Chop(f_of A, f_of B)
          | f_of (Const("True", _)) = True
          | f_of (Const("False", _)) = False
          | f_of (Const("point", _)) = Empty
          | f_of (Const("box", _) $ A) = Box(f_of A)
          | f_of (Const("dia", _) $ A) = Diamond(f_of A)
          | f_of (Const("pr", _) $ A) = Pred(s_of A)
          | f_of (Const("fb", _) $ A $ (Const("pr", _) $ P)) =
            f_of(c_not $ (c_dia $ (c_chop $ A $ (c_pr $ (c_NOT $ P)))))
    in
        f_of t
    end;

fun term_of f =
    let open Formula
        fun t_f f =
            case f of
                True => c_True
              | False => c_False
              | Empty => c_point
              | Box(f1) => c_box $ (t_f f1)
```

```
            | Diamond(f1) => c_dia $ (t_f f1)
            | Neg(f1) => c_not $ (t_f f1)
            | Chop(f1,f2) => c_chop $ (t_f f1) $ (t_f f2)
            | And(f1,f2) => c_conj $ (t_f f1) $ (t_f f2)
            | Or(f1,f2) => c_disj $ (t_f f1) $ (t_f f2)
            | Iff(f1,f2) => c_iff $ (t_f f1) $ (t_f f2)
            | Imp(f1,f2) => c_imp $ (t_f f1) $ (t_f f2)
            | Pred(s) => c_pr $ (t_s s)
        and t_s s =
            case s of
                Id(st) => Var((st, 0), soT)
              | tt => c_tt
              | ff => c_ff
              | sNeg(f1) => c_NOT $ (t_s f1)
              | sAnd(f1,f2) => c_AND $ (t_s f1) $ (t_s f2)
              | sOr(f1,f2) => c_OR $ (t_s f1) $ (t_s f2)
              | sIff(f1,f2) => c_IFF $ (t_s f1) $ (t_s f2)
              | sImp(f1,f2) => c_IMPLIES $ (t_s f1) $ (t_s f2)
    in
        t_f f
    end;

(*
fun dense_valid_fmla fmla =
    case Valid.dense_valid fmla of
        Valid.Valid => fmla
      | Valid.Unsatisfiable => Formula.Neg(fmla)
      | Valid.Counter_example(E) => Formula.Imp(E, Formula.Neg(fmla));

fun TDC_oracle (sign, TDCOracleExn fmla) =
    c_Trueprop $ (sequence []) $ (sequence [term_of (dense_valid_fmla fmla)]);
*)

fun is_dense_valid_fmla fmla =
    case Valid.dense_valid fmla of
        Valid.Valid => true
      | _ => false;

fun TDC_oracle (sign, TDCOracleExn fmla) =
    if is_dense_valid_fmla fmla then
        c_Trueprop $ (sequence []) $ (sequence [term_of fmla])
    else
        raise (TDCOracleExn fmla);
```

A.8 SREAL

```
SREAL = TDC +

types
  sreal
```

```
arities
  sreal :: sterm

consts

  "++"          :: [sreal, sreal] => sreal        (infixl 60)
  "--"          :: [sreal, sreal] => sreal        (infixl 60)
  "---"         :: sreal => sreal                 ("--- _" [80] 80)
  "**"          :: [sreal, sreal] => sreal        (infixl 70)

  "<<"          :: [sreal, sreal] => so           (infixr 50)
  "<<="         :: [sreal, sreal] => so           (infixr 50)
  ">>"          :: [sreal, sreal] => so           (infixr 50)
  ">>="         :: [sreal, sreal] => so           (infixr 50)

  sr            :: rreal => sreal  (* lifting of real *)

  Or            :: sreal => real                  ("Or _")

defs
  sle_def       "x<<=y == x<<y OR x;=y"
  sgt_def       "x>>y == y<<x"
  sge_def       "x>>=y == y<<=x"

rules

  Oadd          "Or(x ++ y) == (Or x) + (Or y)"
  Osub          "Or(x -- y) == (Or x) - (Or y)"
  Omult         "Or(x ** y) == (Or x) * (Or y)"

  Olt           "Oo(x << y) == (Or x) < (Or y)"
  Ole           "Oo(x <<= y) == (Or x) <= (Or y)"
  Ogt           "Oo(x >> y) == (Or x) > (Or y)"
  Oge           "Oo(x >>= y) == (Or x) >= (Or y)"

  Oeq_real      "Oo((x::sreal) ;= y) == (Or x) = (Or y)"
  Osr           "Or(sr(x)) == nr(x)"


  real_induct_chop "[| $G |- H(‘‘), $D;
                      !! X . $G |- H(X) --> H(X|(EX x . ‘s;=sr(x)‘^X)), $D
                   |] ==> $G |- H(True), $D"
  real_chop_induct "[|$G |- H(‘‘), $D;
                      !! X . $G |- H(X) --> H(X|(EX x . X^‘s;=sr(x)‘)), $D
                   |] ==> $G |- H(True), $D"
end
```

A.9 EDC

```
EDC = SREAL +

consts

  (** Operators *)

  intg          :: sreal => real
  bdot          :: sreal => real                  ("b. _")
  edot          :: sreal => real                  ("e. _")

  (** Continuous state recogniser *)

  CNT           :: sreal => prop                  ("CNT _")

rules
  integration "[|RT v1; RT v2|] ==>
               |- intg(s)=v1 ^ intg(s)=v2 --> intg(s) = v1+v2"

  values_b    "RT v ==> |- (b. s)=v <-> (b. s)=v ^ True"
  values_e    "RT v ==> |- (e. s)=v <-> True^ (e. s)=v"

  continuity  "[| CNT cs; RT vb; RT ve |] ==>
               |- (e. cs) = ve ^ (b. cs) = vb --> ve=vb"

  continuity_point "CNT cs ==> |- len=0 --> (e. cs) = (b. cs)"

  mean_inc    "[| CNT cs; RT v; RT vb; RT ve |] ==>
               |- (b. cs)=vb & (e. cs)=ve & vb<=v & v<=ve --> (e. cs)=v ^ True"
  mean_dec    "[| CNT cs; RT v; RT vb; RT ve |] ==>
               |- (b. cs)=vb & (e. cs)=ve & ve<=v & v<=vb --> (e. cs)=v ^ True"

  P_le        "|- `s<<=sr(v)` --> (b. s)<=nr(v) & (e. s)<=nr(v)"
  P_ge        "|- `s>>=sr(v)` --> (b. s)>=nr(v) & (e. s)>=nr(v)"

  P_lt        "|- `s<<sr(v)` --> ((e. s)<nr(v))^((b. s)<nr(v))"
  P_gt        "|- `s>>sr(v)` --> ((e. s)>nr(v))^((b. s)>nr(v))"

  b_lt        "|- (b. s)<nr(v) & len>0 --> `s<<sr(v)`^True"
  b_gt        "|- (b. s)>nr(v) & len>0 --> `s>>sr(v)`^True"
  e_lt        "|- (e. s)<nr(v) & len>0 --> True^`s<<sr(v)`"
  e_gt        "|- (e. s)>nr(v) & len>0 --> True^`s>>sr(v)`"

  P_eq_intg   "|- `s;=sr(v)` --> intg(s)=nr(v)*len"
  P_lt_intg   "|- `s<<sr(v)` --> intg(s)<nr(v)*len"
  P_le_intg   "|- `s<<=sr(v)` --> intg(s)<=nr(v)*len"
  P_gt_intg   "|- `s>>sr(v)` --> intg(s)>nr(v)*len"
  P_ge_intg   "|- `s>>=sr(v)` --> intg(s)>=nr(v)*len"

  point_intg  "|- len=0 --> intg(s)=0"
```

```
  mean_gt     "[| CNT cs; RT vb; RT ve |] ==>
               |- len>0 & (b. cs)=vb & (e. cs)=ve & vb<=nr(v) & nr(v)<ve
                   --> True^((b. cs)=nr(v) & `cs>>sr(v)`)"
  mean_lt     "[| CNT cs; RT vb; RT ve |] ==>
               |- len>0 & (b. cs)=vb & (e. cs)=ve & vb>=nr(v) & nr(v)>ve
                   --> True^((b. cs)=nr(v) & `cs<<sr(v)`)"

  (** Continuity *)

  CNTsr          "CNT(sr(x))"
  CNTadd         "[| CNT x; CNT y|] ==> CNT(x ++ y)"
  CNTsub         "[| CNT x; CNT y|] ==> CNT(x -- y)"
  CNTmult        "[| CNT x; CNT y|] ==> CNT(x ** y)"

end
```

A.10 FB

```
FB= TDCOracle +

consts

  fb            :: [o, o] => o              ("_ --->/ _" [26,25] 25)
  lt            :: [o, real, o] => o        ("_ -= _ ->/ _" [26,50,25] 25)
  ut            :: [o, real, o] => o        ("_ -<= _ ->/ _" [26,50,25] 25)

rules

  (* Definitions *)

  fb_def         "A--->B == ALL r. []((A & len=r)^(len>0) --> (len=r)^B^True)"
  lt_def         "A-=t->B == (A & len=t)--->B"
  ut_def         "A-<=t->B == (A & len<=t)--->B"

end
```

Appendix B

Steam Boiler Source Code

This appendix contains listings of the Isabelle theories forming the encoding of Steam Boiler specification. Proofs scripts of derived rules and definitions of tactic and tacticals are left out for lack of space.

```
SB_COMMIT5
    |
    SB
    |
  SB_TLT
    |
  SB_DEFS
```

Figure B.1: Hierarchy of Steam Boiler theories

B.1 SB_DEFS

```
SB_DEFS = SREAL  +

consts

  abs      :: [real, real] => real
(*  rabs     :: [rreal, rreal] => rreal *)
  sabs     :: [sreal, sreal] => sreal

  max      :: [real, real] => real
(*  rmax     :: [rreal, rreal] => rreal *)

  is_intv  :: [real, real] => o
  lower    :: [real, real, real, real] => o
```

```
  higher   :: [real, real, real, real] => o
  subintv  :: [real, real, real, real] => o
  psubintv :: [real, real, real, real] => o

  member   :: [real, real, real] => o
(*  rmember  :: [rreal, rreal, rreal] => ro*)
  smember  :: [sreal, sreal, sreal] => so

defs
  is_intv_def  "is_intv(l, h) == 0 <= l & l <= h"
  lower_def    "lower(la, ha, lb, hb) == la < lb & ha < hb"
  higher_def   "higher(la, ha, lb, hb) == la > lb & ha > hb"
  subintv_def  "subintv(la, ha, lb, hb) == la >= lb & ha <= hb"
  psubintv_def "psubintv(la, ha, lb, hb) ==
                la > lb & ha <= hb | la >= lb & ha < hb"

  member_def   "member(x, l, h) == l <= x & x <= h"

rules
  abs_ax    "|- (l > h --> abs(l, h)=l-h) & (l <= h --> abs(l, h)=h-l)"
  max_ax    "|- (a > b --> max(a, b)=a) & (a <= b --> max(a, b)=b)"

  abs_cong  "|- l=l' & h=h' --> abs(l, h) = abs(l',h')"
  max_cong  "|- a=a' & b=b' --> max(a, b) = max(a',b')"

  Omember   "Oo(smember(x, l, h)) == (member(Or(x),Or(l),Or(h)))"
  Oabs      "Or(sabs(l, h)) == abs(Or(l),Or(h))"

(*  nomember  "member(nr(x),nr(l),nr(h))==no(rmember(x,l,h))"
  nrabs     "abs(nr(l),nr(h))==nr(rabs(l,h))"
  nrmax     "max(nr(l),nr(h))==nr(rmax(l,h))"
*)
  RTabs     "[|RT(l); RT(h)|] ==> RT(abs(l, h))"
  RTmax     "[|RT(a); RT(b)|] ==> RT(max(a, b))"
  RTmember  "[|RT(x); RT(l); RT(h)|] ==> RT(member(x, l, h))"

end
```

B.2 SB_TLT

```
SB_TLT = SB_DEFS +

consts

  lN   :: rreal
  hN   :: rreal
  lNN  :: rreal
  hNN  :: rreal

  LHS1 :: [real, o, o, o, real, o, o, o, real, real] => o
```

```
  LHS2 :: [real, o, o, o, real, o, o, o, real, real] => o
  LHS3 :: [real, o, o, o, real, o, o, o, real, real] => o
  LHS4 :: [real, o, o, o, real, o, o, o, real, real] => o
  LHS5 :: [real, o, o, o, real, o, o, o, real, real] => o
  R    :: [real, o, o, o, real, o, o, o, real, real] => o
  I    :: [real, o, o, o, real, real] => o

  (* Parameter *)
  C     :: rreal (* Maximum waterlevel *)

defs

  LHS1_def "LHS1(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj', hadj') ==
            a'>a & ~ pRd & ~ vCmd & ~ pCmd & higher(ladj', hadj', nr(lN), nr(hN))"

  LHS2_def "LHS2(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj', hadj') ==
            a'>a & ~ pRd & subintv(ladj', hadj', nr(lNN), nr(hNN))"

  LHS3_def "LHS3(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj', hadj') ==
            a'>a & ~ pRd & ~ vCmd & ~ pCmd & subintv(ladj', hadj', nr(lN), nr(hN))"

  LHS4_def "LHS4(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj', hadj') ==
            a'>a & ~ vCmd & ~ pCmd & lower(ladj', hadj', nr(lN), nr(hN))"

  LHS5_def "LHS5(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj', hadj') ==
            a'>a & pRd & higher(ladj', hadj', nr(lN), nr(hN))"

  I_def  "I(a, pRd, vCmd, pCmd, ladj, hadj) ==
          a = 0 & ladj = 0 & hadj = nr(C) & ~pRd & ~pCmd & ~vCmd"

  R_def  "R(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj', hadj') ==
          (LHS1(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj', hadj') --> vCmd') &
          (LHS2(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj', hadj') -->
              ~ vCmd' & ~ pCmd') &
          (LHS3(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj', hadj') --> pRd') &
          (LHS4(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj', hadj') --> pCmd') &
  (LHS5(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj', hadj') --> ~ pCmd') &
  (~ (vCmd' <-> vCmd) -->
     LHS1(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj', hadj') |
     LHS2(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj', hadj')) &
  (~ (pCmd' <-> pCmd) -->
     LHS2(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj', hadj') |
     LHS4(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj', hadj') |
     LHS5(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj', hadj')) &
  (~ (pRd' <-> pRd) -->
     LHS3(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj', hadj'))"

rules
  N_intv  "|- is_intv(nr(lN), nr(hN))"
  NN_intv "|- is_intv(nr(lNN), nr(hNN))"
  NN_N    "|- psubintv(nr(lNN), nr(hNN), nr(lN), nr(hN))"
```

```
  RTI "[|RT(n); RT(pRd); RT(vCmd); RT(pCmd); RT(ladj); RT(hadj)|]
       ==> RT(I(n, pRd, vCmd, pCmd, ladj, hadj))"

  RTR "[|RT(n); RT(pRd); RT(vCmd); RT(pCmd);
         RT(n'); RT(pRd'); RT(vCmd'); RT(pCmd'); RT(ladj'); RT(hadj')|]
       ==> RT(R(n, pRd, vCmd, pCmd, n', pRd', vCmd', pCmd', ladj', hadj'))"

 end
```

B.3 SB

```
SB = EDC + FB + SB_TLT +

consts

  (* Parameters *)

  tau   :: rreal

  P     :: rreal (* Maximum pump throughput *)
  D     :: rreal (* Maximum valve throughput *)
  W     :: rreal (* Maximum steam throughput *)

  lM    :: rreal
  hM    :: rreal

  (* Environment states *)

  q :: sreal (* Water level *)
  v :: sreal (* Steam thoughput *)
  p :: sreal (* Pump throughput *)
  d :: sreal (* Valve throughput *)

  (* Controller state *)

  ready :: so

  sa :: sreal
  sladj :: sreal
  shadj :: sreal
  spRd :: so
  spCmd :: so
  svCmd :: so

  (* Lifted invariant *)

  sI :: [sreal, so, so, so, sreal, sreal] => so
(*   rI :: [rreal, ro, ro, ro, rreal, rreal] => ro *)
```

```
  (* Abbreviations *)
  AC4 :: o
  Sem1 :: o
  Init :: o

  eq_s  :: [rreal, ro, ro, ro, rreal, rreal] => so
  eq_ss :: [rreal, ro, ro, ro, rreal, rreal, rreal, ro, ro, ro, rreal, rreal] => o

defs
  AC4_def   "AC4 == ‘‘ | (‘sa;=sr(0)‘ & len < nr(tau)) |
                   ((‘sa;=sr(0)‘ & len = nr(tau))^True)"

  Sem1_def  "Sem1 == ‘‘ | ‘sI(sa,spRd,svCmd,spCmd,sladj,shadj)‘^True"

  Init_def  "Init == Sem1 & AC4"

  eq_s_def  "eq_s(n,pRd,vCmd,pCmd,ladj,hadj)==
             sa;=sr(n) AND spRd;=so(pRd) AND svCmd;=so(vCmd) AND
             spCmd;=so(pCmd) AND sladj;=sr(ladj) AND shadj;=sr(hadj)"

  eq_ss_def "eq_ss(n,pRd,vCmd,pCmd,ladj,hadj,n’,pRd’,vCmd’,pCmd’,ladj’,hadj’)==
             no(n:=n’ & pRd:=pRd’ & vCmd:=vCmd’ & pCmd:=pCmd’ &
                ladj:=ladj’ & hadj:=hadj’)"

rules
  taupos "|- nr(tau) > 0"
  Cpos   "|- nr(C) > 0"
  Ppos   "|- nr(P) > 0"
  Dpos   "|- nr(D) > 0"
  Wpos   "|- nr(W) > 0"

  M_intv "|- is_intv(nr(lM), nr(hM))"
  N_M    "|- psubintv(nr(lN), nr(hN), nr(lM), nr(hM))"

  q_range "|- ‘‘ | ‘q>>=sr(0) AND q<<=sr(C)‘"
  v_range "|- ‘‘ | ‘v>>=sr(0) AND v<<=sr(W)‘"
  p_range "|- ‘‘ | ‘p>>=sr(0) AND p<<=sr(P)‘"
  d_range "|- ‘‘ | ‘d>>=sr(0) AND d<<=sr(D)‘"

  Mass "|- (e. q) - (b. q) = intg(p) - intg(v) - intg(d)"


  (* Program clock *)
  AC1 "|- ‘sa;=sr(n)‘ ---> ‘sa;=sr(n) OR sa;=sr(n+1)‘"
  AC2 "|- ‘sa;=sr(n)‘ -= nr(tau) -> ‘NOT sa;=sr(n)‘"
  AC3 "|- (‘NOT sa;=sr(n)‘^‘sa;=sr(n)‘) -<= nr(tau) -> ‘sa;=sr(n)‘"

  (* Clock syncronisation *)
  Sync_pRd "|- ‘sa;=sr(n) AND spRd;=so(pRd)‘ --->
                 ‘NOT sa;=sr(n) OR spRd;=so(pRd)‘"
```

```
  Sync_vCmd "|- `sa;=sr(n) AND svCmd;=so(vCmd)` --->
                  `NOT sa;=sr(n) OR svCmd;=so(vCmd)`"
  Sync_pCmd "|- `sa;=sr(n) AND spCmd;=so(pCmd)` --->
                  `NOT sa;=sr(n) OR spCmd;=so(pCmd)`"
  Sync_ladj "|-`sa;=sr(n) AND sladj;=sr(lV)` --->
                `NOT sa;=sr(n) OR sladj;=sr(lV)`"
  Sync_hadj "|-`sa;=sr(n) AND shadj;=sr(hV)` --->
                `NOT sa;=sr(n) OR shadj;=sr(hV)`"

  WLS1 "|- ((e. q)=nr(q0) & `sa;=sr(n)`) --->
              `sa;=sr(n) OR smember(sr(q0), sladj, shadj)`"

  PA1  "|- `spCmd` -= nr(tau) -> `p;=sr(P)`"
  PA2  "|- `NOT spCmd` -= nr(tau) -> `p;=sr(0)`"

  VA1  "|- `svCmd` -= nr(tau) -> `d;=sr(D)`"
  VA2  "|- `NOT svCmd` -= nr(tau) -> `d;=sr(0)`"

  RdyA1 "|- `` | `ready IFF spRd`"


  TLTlift "[| Oo(sInv(sa, spRd, svCmd, spCmd, sladj, shadj)) ==
                Inv(Or sa, Oo spRd, Oo svCmd, Oo spCmd, Or sladj, Or shadj);
             |- (ALL n pRd vCmd pCmd ladj hadj n' pRd' vCmd' pCmd' ladj' hadj' .
                 (I(n,pRd,vCmd,pCmd,ladj,hadj)
                     --> Inv(n,pRd,vCmd,pCmd,ladj,hadj)) &
                 (R(n,pRd,vCmd,pCmd,n',pRd',vCmd',pCmd',ladj',hadj') &
                  Inv(n,pRd,vCmd,pCmd,ladj,hadj)
                     --> Inv(n',pRd',vCmd',pCmd',ladj',hadj')))
           |] ==> |- [](`` | `sInv(sa,spRd,svCmd,spCmd,sladj,shadj)`)"

  Sem2    "|- `eq_s(n,pRd,vCmd,pCmd,ladj,hadj)`^
             `eq_s(n',pRd',vCmd',pCmd',ladj',hadj')` -->
             R(nr(n),no(pRd),no(vCmd),no(pCmd),
               nr(n'),no(pRd'),no(vCmd'),no(pCmd'),nr(ladj'),nr(hadj'))"

  OI  "Oo(sI(a, pRd, vCmd, pCmd, ladj, hadj)) ==
        I(Or(a),Oo(pRd),Oo(vCmd),Oo(pCmd),Or(ladj),Or(hadj))"

(*  noI  "I(nr(a),no(pRd),no(vCmd),no(pCmd),nr(ladj),nr(hadj))==
       no(rI(a,pRd,vCmd,pCmd,ladj,hadj))" *)

  RTeq_ss "[|RT(n); RT(pRd); RT(vCmd); RT(pCmd); RT(ladj); RT(hadj);
             RT(n'); RT(pRd'); RT(vCmd'); RT(pCmd'); RT(ladj'); RT(hadj')|]
          ==> RT(eq_ss(n, pRd, vCmd, pCmd, ladj, hadj,
                       n', pRd', vCmd', pCmd', ladj', hadj'))"



  (* Continuous states *)
  CNTq "CNT q"
```

```
  CNTv "CNT v"
  CNTp "CNT p"
  CNTd "CNT d"

end
```

B.4 SB_COMMIT5

```
SB_COMMIT5  =  SB  +

consts

  epsilon :: rreal

rules

  (* Assumptions under which Commit5 is proved *)
  WLSTol "|- `` | `(sabs(sladj, shadj)) << sr(two*epsilon)`"

  epsilonpos "|- nr(epsilon) > 0"

  interval_assump "|-
    nr(hM - hN) > 2*nr(tau)*max(nr(P), nr(D)) + 2*nr(epsilon) &
    nr(hN - hNN) > nr(tau)*max(nr(P), nr(D)) + 2*nr(epsilon) &
    nr(hN - lNN) > nr(tau)*max(nr(P), nr(D)) + 4*nr(epsilon) &
    nr(lNN - lN) > nr(tau)*max(nr(P), nr(D)) + 2*nr(epsilon) &
    nr(lN - lM) > 2*nr(tau)*max(max(nr(P), nr(D)), nr(W)) + 2*nr(epsilon)"

  P_assump "|- no(P:>=W)"

  (* Should be moved to RA.thy *)
  both_sides_times_pos_gt1 "|- pz > 0 --> x * pz > y * pz <-> x > y"
  both_sides_times_pos_gt2 "|- pz > 0 --> pz * x > pz * y <-> x > y"

  both_sides_times_pos_ge1 "|- pz > 0 --> x * pz >= y * pz <-> x >= y"
  both_sides_times_pos_ge2 "|- pz > 0 --> pz * x >= pz * y <-> x >= y"

  both_sides_times_pos_lt1 "|- pz > 0 --> x * pz < y * pz <-> x < y"
  both_sides_times_pos_lt2 "|- pz > 0 --> pz * x < pz * y <-> x < y"

  both_sides_times_pos_le1 "|- pz > 0 --> x * pz <= y * pz <-> x <= y"
  both_sides_times_pos_le2 "|- pz > 0 --> pz * x <= pz * y <-> x <= y"

end

ML
```

Appendix C

Isabelle/DC Theorems

This appendix contains what is basically a dump of Isabelles internal theorem data base. The dump provides evidence that the listed theorems have in fact been derived using Isabelle.

The first section lists the ML code used to extract the dump. Following sections lists theorems of particular theories. Lists of theorems are sorted alphabetically by name. Only derived theorems are listed to avoid redundancy and to save space. Axioms are declared in the Isabelle theories listed in Appendix A.

C.1 Extraction code

```
(*Replace outermost quantified variable by Free
    Could clash with Frees already present.*)
fun forall_elim_free th =
    let val {prop,sign,...} = rep_thm th
    in case prop of
          Const("all",_) $ Abs(a,T,_) =>
              forall_elim (cterm_of sign (Free(a, T)))  th
        | _ => raise THM("forall_elim_free", 0, [th])
    end;

(*Repeat forall_elim_free until all outer quantifiers are removed*)
fun forall_elim_frees th =
    forall_elim_frees (forall_elim_free th)
        handle THM _ => th;

fun forall_intr_vars thm =
  let
      val vars = term_vars (Thm.term_of (cprop_of thm))
      val sign = sign_of_thm thm
  in
      (forall_intr_list
      (map (fn t => cterm_of sign t) vars )
```

```
      thm)
  end;

(* Same as standard by unknowns are replaced by free variables
   (without the ? sign which we dont want in the dump *)
val standard_frees = forall_elim_frees o forall_intr_vars o standard;

(* Formats a named theorem with a fixwidth field for the name followed
   by the formated theorem *)
fun pretty_named_thm nm_width (nm, thm) =
  let
    val s_nm = (String.size nm)
    val n = nm_width - s_nm
  in
      if n < 2 then
          Pretty.blk ((nm_width),
                      [(Pretty.str nm),(Pretty.fbrk),(pretty_thm thm)])
      else
          Pretty.blk ((nm_width),
                      [(Pretty.str nm),(Pretty.brk n),(pretty_thm thm)])
  end;


fun string_of_thms thms =
    foldl
    (fn (s, (n, t)) =>
     s ^
     (Pretty.string_of(pretty_named_thm 16 (n, standard_frees t))) ^
     "\n")
    ("", thms);

fun dump_thy_thms thy =
    let
        val thy_nm = List.last (Sign.stamp_names_of(sign_of thy))
        val file_nm = "/home/sh/thesis/"^thy_nm^"_thms"
        val out_strm = TextIO.openOut file_nm
        val axs = axioms_of thy
        val derived_thms =
            filter (fn (nm,_) => not (exists (fn (n,_) => nm=n) axs)) (thms_of thy)
    in
        print_mode := ""::(!print_mode);
        TextIO.outputSubstr(out_strm,
                            Substring.all (string_of_thms derived_thms));
        TextIO.closeOut out_strm;
        print_mode := tl (!print_mode)
    end;


val isadc_thys = [Sequence.thy, PL.thy, RA.thy, RAOracle.thy, IL.thy, TDC.thy,
                  TDCOracle.thy, SREAL.thy, EDC.thy, FB.thy];
```

```
val sb_thys = [SB_DEFS.thy, SB_TLT.thy, SB.thy, SB_COMMIT5.thy];

fun dump_em () =
    (Pretty.setmargin 80;
     app dump_thy_thms (isadc_thys@sb_thys));
```

C.2 PL

```
IL11lem2          |- (ALL x. A(x) --> B) --> (EX x. A(x)) --> B
K1                |- A --> B --> A
K2                |- (A --> B --> C) --> (A --> B) --> A --> C
K3                |- (~ A --> ~ B) --> B --> A
K5CF             CF A(x) ==>  |- (ALL x. A(x)) --> A(x)
K5RT             RT x ==>  |- (ALL x. A(x)) --> A(x)
K6                |- (ALL x. A --> B(x)) --> A --> (ALL x. B(x))
P_iff_F           |- ~ P ==>  |- P <-> False
P_iff_T           |- P ==>  |- P <-> True
TrueR            $H |- $E, True, $F
allLCF_thin      [| $H, P(x), $G |- $E; CF P(x) |] ==> $H, ALL x. P(x), $G |- $E
allLRT_thin      [| $H, P(x), $G |- $E; RT x |] ==> $H, ALL x. P(x), $G |- $E
all_ex            |- (ALL x. A(x)) <-> ~ (EX x. ~ A(x))
comm_conj_equiv_thinL
                 [|  |- A & A' <-> B; $I, B, $H, $G |- $E |]
                 ==> $I, A', $H, A, $G |- $E
comm_conj_imply_thinL
                 [|  |- A & A' --> B; $I, B, $H, $G |- $E |]
                 ==> $I, A', $H, A, $G |- $E
conL             $H, P, $G, P |- $E ==> $H, P, $G |- $E
conR             $H |- $E, P, $F, P ==> $H |- $E, P, $F
cond_comm_conj_imply_thinL
                 [| C |- A & A' --> B; $I, B, $H, $G |- $E;
                    $I, $H, $G |- C, $E |]
                 ==> $I, A', $H, A, $G |- $E
cond_conj_imply_thinL
                 [| C |- A & A' --> B; $I, B, $H, $G |- $E;
                    $I, $H, $G |- C, $E |]
                 ==> $I, A, $H, A', $G |- $E
cond_equiv_imply
                 C |- A <-> B ==> C |- A --> B
cond_imply_thinL
                 [| C |- A --> B; $H, B, $G |- $E; $H, $G |- C, $E |]
                 ==> $H, A, $G |- $E
cond_lemma       [| A |- B; $G |- A, $E; B, $G |- $E |] ==> $G |- $E
cond_lemma2      [| $G |- A, $E; B, $G |- $E; A |- B |] ==> $G |- $E
conjLE           $G, A & B |- $E ==> $G, A, B |- $E
conjR_first      [| $G |- P, $E; $G |- Q, $E |] ==> $G |- P & Q, $E
conj_assoc        |- (A & B) & C <-> A & B & C
conj_comm         |- A & B <-> B & A
conj_commute      |- P & Q <-> Q & P
```

```
conj_disj_distribL
                 |- P & (Q | R) <-> P & Q | P & R
conj_disj_distribR
                 |- (P | Q) & R <-> P & R | Q & R
conj_equiv_thinL
                [|  |- A & A' <-> B; $I, B, $H, $G |- $E |]
                ==> $I, A, $H, A', $G |- $E
conj_imply_thinL
                [|  |- A & A' --> B; $I, B, $H, $G |- $E |]
                ==> $I, A, $H, A', $G |- $E
conj_left_comm   |- A & B & C <-> B & A & C
conj_left_commute
                 |- P & Q & R <-> Q & P & R
conjunct1        |- P & Q ==>  |- P
conjunct2        |- P & Q ==>  |- Q
contL           $H, P, P, $G |- $E ==> $H, P, $G |- $E
contR           $H |- $E, P, P, $F ==> $H |- $E, P, $F
de_Morgan_conj   |- ~ (P & Q) <-> ~ P | ~ Q
de_Morgan_disj   |- ~ (P | Q) <-> ~ P & ~ Q
disjL_first     [| P, $G |- $E; Q, $G |- $E |] ==> P | Q, $G |- $E
disjRE          $G |- $E, A | B ==> $G |- $E, A, B
disj_commute     |- P | Q <-> Q | P
disj_conj_distribL
                 |- P | Q & R <-> (P | Q) & (P | R)
disj_conj_distribR
                 |- P & Q | R <-> (P | R) & (Q | R)
disj_left_commute
                 |- P | Q | R <-> Q | P | R
eq_iffL         $H, A <-> B, $G |- $E ==> $H, A = B, $G |- $E
eq_iffR         $G |- $E, A <-> B, $F ==> $G |- $E, A = B, $F
eq_repl         $H, a = b, $G |- $E, f(a) = f(b), $F
equivL          [| $H, $G |- A <-> B, $E; $H, B, $G, A |- $E |]
                ==> $H, A, $G |- $E
equivR          [| $G |- A <-> B, $E, $F; $G |- $E, B, $F, A |]
                ==> $G |- $E, A, $F
equiv_comm_disj_thinR
                [|  |- A <-> B | B'; $H |- $E, A, $F, $G |]
                ==> $H |- $E, B', $F, B, $G
equiv_cond       |- C --> A <-> B ==> C |- A <-> B
equiv_disj_thinR
                [|  |- A <-> B | B'; $H |- $E, A, $F, $G |]
                ==> $H |- $E, B, $F, B', $G
equiv_imply      |- A <-> B ==>  |- A --> B
equiv_thinL     [|  |- A <-> B; $H, B, $G |- $E |] ==> $H, A, $G |- $E
equiv_thinR     [|  |- A <-> B; $G |- $E, B, $F |] ==> $G |- $E, A, $F
exLexR          (!!x. $H, A(x), $G |- $E, B(x), $F)
                ==> $H, EX x. A(x), $G |- $E, EX x. B(x), $F
exRCF_thin      [| $H |- $E, P(x), $F; CF P(x) |] ==> $H |- $E, EX x. P(x), $F
exRRT_thin      [| $H |- $E, P(x), $F; RT x |] ==> $H |- $E, EX x. P(x), $F
ex_all           |- (EX x. A(x)) <-> ~ (ALL x. ~ A(x))
exchL           $H, Q, P, $G |- $E ==> $H, P, Q, $G |- $E
```

```
exchR           $H |- $E, Q, P, $F ==> $H |- $E, P, Q, $F
firstL          P, $H, $G |- $E ==> $H, P, $G |- $E
firstR          $G |- P, $E, $F ==> $G |- $E, P, $F
hej             $S |- $R ==> $H, $S, $G |- $E, $R, $F
iffL            [| $H, $G |- $E, P, Q; $H, Q, P, $G |- $E |]
                ==> $H, P <-> Q, $G |- $E
iffR            [| $H, P |- $E, Q, $F; $H, Q |- $E, P, $F |]
                ==> $H |- $E, P <-> Q, $F
iff_refl         |- A <-> A
iff_sym          |- A <-> B ==>  |- B <-> A
iff_trans       [|  |- A <-> B;  |- B <-> C |] ==>  |- A <-> C
impL_first      [| $G |- P, $E; Q, $G |- $E |] ==> P --> Q, $G |- $E
impRE           $G |- A --> B, $E ==> A, $G |- B, $E
imp_conj         |- (P & Q --> R) <-> P --> Q --> R
imp_conj_distrib
                 |- (P --> Q & R) <-> (P --> Q) & (P --> R)
imp_disj         |- (P | Q --> R) <-> (P --> R) & (Q --> R)
imp_disj1        |- (P --> Q) | R <-> P --> Q | R
imp_disj2        |- Q | (P --> R) <-> P --> Q | R
imply_comm_disj_thinR
                [|  |- B --> A | A'; $H |- $E, B, $F, $G |]
                ==> $H |- $E, A', $F, A, $G
imply_cond       |- C --> A --> B ==> C |- A --> B
imply_disj_thinR
                [|  |- B --> A | A'; $H |- $E, B, $F, $G |]
                ==> $H |- $E, A, $F, A', $G
imply_thinL     [|  |- A --> B; $H, B, $G |- $E |] ==> $H, A, $G |- $E
imply_thinR     [|  |- B --> A; $G |- $E, B, $F |] ==> $G |- $E, A, $F
lemma           [|  |- A; A, $G |- $E |] ==> $G |- $E
lemma2          [| A, $G |- $E;  |- A |] ==> $G |- $E
mp              [|  |- P --> Q;  |- P |] ==>  |- Q
notL_first      $G |- P, $E ==> ~ P, $G |- $E
notR_first      P, $G |- $E ==> $G |- ~ P, $E
not_iff          |- ~ (P <-> Q) <-> P <-> ~ Q
simpL           [| A |- B; $H, B, $G |- $E |] ==> $H, A, $G |- $E
spec            [|  |- ALL x. P(x); RT x |] ==>  |- P(x)
subst            |- a = b ==> $H, A(a), $G |- $E, A(b), $F
sym             $H |- $E, a = b, $F ==> $H |- $E, b = a, $F
symL            $H, $G, B = A |- $E ==> $H, A = B, $G |- $E
thinL           $H, $G |- $E ==> $H, P, $G |- $E
thinR           $H |- $E, $F ==> $H |- $E, P, $F
thin_doubleL    $F, P, $G, $H |- $E ==> $F, P, $G, P, $H |- $E
thin_doubleR    $H |- $E, P, $F, $G ==> $H |- $E, P, $F, P, $G
trans           [| $H |- $E, a = b, $F; $H |- $E, b = c, $F |]
                ==> $H |- $E, a = c, $F
```

C.3 RA

```
K4real           |- (ALL x. A) --> A
```

C.4 IL

```
G               (!!x.  |- A(x)) ==>  |- ALL x. A(x)
IL1              |- [](A --> B) --> []A --> []B
IL11             |- ALL x. len = x --> A ==>  |- A
IL11lem1         |- EX x. len = x
IL12             |- (EX x. A(x) ^ B(x)) --> (EX x. A(x)) ^ (EX x. B(x))
IL13a            |- A ^ False <-> False
IL13b            |- False ^ A <-> False
IL14a            |- (A | B) ^ C <-> A ^ C | B ^ C
IL14b            |- A ^ (B | C) <-> A ^ B | A ^ C
IL15a           RT x ==>  |- (len = x & A) ^ B --> ~ (len = x & ~ A) ^ B
IL15b           RT x ==>  |- A ^ (len = x & B) --> ~ A ^ (len = x & ~ B)
IL16a           RT x
                ==>  |- (A & len = x) ^ B & (A' & len = x) ^ B' -->
                        (A & A' & len = x) ^ (B & B')
IL16alem1       RT x
                ==>  |- (A & len = x) ^ B & (A' & len = x) ^ B -->
                        (A & A' & len = x) ^ B
IL16alem2       RT x
                ==>  |- (A & len = x) ^ B & len = x ^ B' -->
                        (A & len = x) ^ (B & B')
IL16b           RT x
                ==>  |- A ^ (B & len = x) & A' ^ (B' & len = x) -->
                        (A & A') ^ (B & B' & len = x)
IL16blem1       RT x
                ==>  |- A ^ (B & len = x) & A ^ (B' & len = x) -->
                        A ^ (B & B' & len = x)
IL16blem2       RT x
                ==>  |- A ^ (B & len = x) & A' ^ len = x -->
                        (A & A') ^ (B & len = x)
IL17a            |- A ^ len = 0 --> A
IL17b            |- len = 0 ^ A --> A
IL18a            |- []A & B ^ C --> (A & B) ^ C
IL18b            |- []A & B ^ C --> B ^ (A & C)
IL2              |- []A --> A
IL20             |- [](A & B) <-> []A & []B
IL21             |- []A | []B --> [](A | B)
IL3              |- []A --> [][]A
IL4              |- A ==>  |- []A
IL5              |- []A --> ~ (~ A) ^ B
IL6              |- []A --> B ==>  |- []A --> []B
IL7              |- [](A --> A') --> A ^ B --> A' ^ B
IL8              |- [](A --> B) --> []([]A --> []B)
IL9             RT A ==>  |- A --> []A
L1bC             |- ALL x. A(x) ^ len = x --> ~ (~ A(x)) ^ len = x
M1               |- A --> B ==>  |- A ^ C --> B ^ C
M2               |- A --> B ==>  |- C ^ A --> C ^ B
MP              [|  |- A;  |- A --> B |] ==>  |- B
N1               |- A ==>  |- ~ (~ A) ^ B
N2               |- A ==>  |- ~ B ^ (~ A)
```

```
Nlem1             |- [](B --> True)
Nlem2             |- A ==>  |- [](~ A --> True ^ (~ A))
Nlem3            True ^ (~ A) ^ True |-  ==> (True ^ (~ A)) ^ True |-
Nlem4             |- A ==>  |- [](~ A --> (~ A) ^ True)
QaCF             CF A(x) ==>  |- (ALL x. A(x)) --> A(x)
QaRT             RT x ==>  |- (ALL x. A(x)) --> A(x)
QbCF             CF A(x) ==>  |- A(x) --> (EX x. A(x))
QbRT             RT x ==>  |- A(x) --> (EX x. A(x))
RT_chop          [| RT A; $H, A, $G, A ^ B |- $E |] ==> $H, A ^ B, $G |- $E
RT_chop_equiv    RT R ==>  |- (R & A) ^ B <-> R & A ^ B
RT_chop_thin     [| RT A; $H, A, $G |- $E |] ==> $H, A ^ B, $G |- $E
RT_true_chop_equiv
                 RT R ==>  |- R ^ B <-> R & True ^ B
RaS              RT A ==> A ^ B |- A
RbS              RT A ==> B ^ A |- A
all_box           |- (ALL x. []A(x)) <-> [](ALL x. A(x))
boxL             $E, []P, $F, P |- $G ==> $E, []P, $F |- $G
boxL_thin        $E, P, $F |- $G ==> $E, []P, $F |- $G
boxR_bang         |- A ==> $G |- $E, []A, $F
box_RT           [| RT A; $H, []A, $G, A |- $E |] ==> $H, A, $G |- $E
box_RT_thin      [| RT A; $H, []A, $G |- $E |] ==> $H, A, $G |- $E
case_chop        [| $H, (D & A) ^ B, $G |- $E; $H, (~ D & A) ^ B, $G |- $E |]
                 ==> $H, A ^ B, $G |- $E
chop_RT          [| RT B; $H, B, $G, A ^ B |- $E |] ==> $H, A ^ B, $G |- $E
chop_RT_chop     [| RT B; $H, B, $G, A ^ B ^ C |- $E |]
                 ==> $H, A ^ B ^ C, $G |- $E
chop_RT_chop_thin
                 [| RT B; $H, B, $G |- $E |] ==> $H, A ^ B ^ C, $G |- $E
chop_RT_equiv    RT R ==>  |- A ^ (R & B) <-> R & A ^ B
chop_RT_thin     [| RT B; $H, B, $G |- $E |] ==> $H, A ^ B, $G |- $E
chop_RT_true_equiv
                 RT R ==>  |- A ^ R <-> R & A ^ True
chop_case        [| $H, A ^ (D & B), $G |- $E; $H, A ^ (~ D & B), $G |- $E |]
                 ==> $H, A ^ B, $G |- $E
chop_case_chop   [| $H, A ^ (D & B) ^ C, $G |- $E;
                    $H, A ^ (~ D & B) ^ C, $G |- $E |]
                 ==> $H, A ^ B ^ C, $G |- $E
chop_conj        [| RT x; $I, $H, $G |- [](B --> len = x), $E;
                    $I, $H, $G |- [](B' --> len = x), $E;
                    (A & A') ^ (B & B'), $I, $H, $G |- $E |]
                 ==> $I, A ^ B, $H, A' ^ B', $G |- $E
chop_cut         [| A ^ B |- C; C, $H, A ^ B, $G |- $E |] ==> $H, A ^ B, $G |- $E
chop_disj         |- A ^ (B | C) <-> A ^ B | A ^ C
chop_disjL       [| $H, A ^ B, $G |- $E; $H, A ^ C, $G |- $E |]
                 ==> $H, A ^ (B | C), $G |- $E
chop_disjR       $G |- $E, A ^ B, A ^ C, $F ==> $G |- $E, A ^ (B | C), $F
chop_disj_chop    |- A ^ (B | C) ^ D <-> A ^ B ^ D | A ^ C ^ D
chop_disj_chopL
                 [| $H, A ^ B ^ D, $G |- $E; $H, A ^ C ^ D, $G |- $E |]
                 ==> $H, A ^ (B | C) ^ D, $G |- $E
chop_disj_chopR
```

```
                $G |- $E, A ^ B ^ D, A ^ C ^ D, $F
                ==> $G |- $E, A ^ (B | C) ^ D, $F
chop_equiv_thinL
                [|  |- A ^ A' <-> B; $H, B, $G |- $E |] ==> $H, A ^ A', $G |- $E
chop_equiv_thinR
                [|  |- A ^ A' <-> B; $G |- $E, B, $F |] ==> $G |- $E, A ^ A', $F
chop_exists     $H, EX x. A ^ B(x), $G, A ^ (EX x. B(x)) |- $E
                ==> $H, A ^ (EX x. B(x)), $G |- $E
chop_exists_chop
                $H, EX x. A ^ B(x) ^ C, $G, A ^ (EX x. B(x)) ^ C |- $E
                ==> $H, A ^ (EX x. B(x)) ^ C, $G |- $E
chop_exists_chop_thin
                $H, EX x. A ^ B(x) ^ C, $G |- $E
                ==> $H, A ^ (EX x. B(x)) ^ C, $G |- $E
chop_exists_equiv
                 |- A ^ (EX x. B(x)) <-> (EX x. A ^ B(x))
chop_exists_nonneg_equiv
                 |- A ^ (EX x. x >= 0 & B(x)) <-> (EX x. x >= 0 & A ^ B(x))
chop_exists_nonneg_len_equiv
                 |- A ^ B <-> A ^ (EX x. x >= 0 & len = x & B)
chop_exists_thin
                $H, EX x. A ^ B(x), $G |- $E ==> $H, A ^ (EX x. B(x)), $G |- $E
chop_false      A ^ False |-
chop_lemma      [|  |- D; $H, A ^ (D & B), $G |- $E |] ==> $H, A ^ B, $G |- $E
chop_lemma_chop
                [|  |- D; $H, A ^ (D & B) ^ C, $G |- $E |]
                ==> $H, A ^ B ^ C, $G |- $E
chop_len        (!!x. RT x ==> $H, A ^ (len = x & B), $G, x >= 0 |- $E)
                ==> $H, A ^ B, $G |- $E
chop_len_chop   (!!x. RT x ==> $H, A ^ (len = x & B) ^ C, $G, x >= 0 |- $E)
                ==> $H, A ^ B ^ C, $G |- $E
chop_mono       [| $H, $G |- $E, [](A --> B); $H, C ^ B, $G, C ^ A |- $E |]
                ==> $H, C ^ A, $G |- $E
chop_mono_RT    [| RT R;  |- B --> R; $H, A ^ B, $G, R |- $E |]
                ==> $H, A ^ B, $G |- $E
chop_mono_RT_chop
                [| RT R;  |- B --> R; $H, A ^ B ^ C, $G, R |- $E |]
                ==> $H, A ^ B ^ C, $G |- $E
chop_mono_bang  [|  |- A --> B; $H, C ^ B, $G, C ^ A |- $E |]
                ==> $H, C ^ A, $G |- $E
chop_mono_bang_thin
                [|  |- A --> B; $H, C ^ B, $G |- $E |] ==> $H, C ^ A, $G |- $E
chop_mono_chop  [| $H, $G |- $E, [](A --> B);
                   $H, C ^ B ^ D, $G, C ^ A ^ D |- $E |]
                ==> $H, C ^ A ^ D, $G |- $E
chop_mono_chop_bang
                [|  |- A --> B; $H, C ^ B ^ D, $G, C ^ A ^ D |- $E |]
                ==> $H, C ^ A ^ D, $G |- $E
chop_mono_chop_bang_thin
                [|  |- A --> B; $H, C ^ B ^ D, $G |- $E |]
                ==> $H, C ^ A ^ D, $G |- $E
```

```
chop_mono_chop_first_thin
                [| $H, $G |- [](A --> B), $E; $H, C ^ B ^ D, $G |- $E |]
                ==> $H, C ^ A ^ D, $G |- $E
chop_mono_chop_thin
                [| $H, $G |- $E, [](A --> B); $H, C ^ B ^ D, $G |- $E |]
                ==> $H, C ^ A ^ D, $G |- $E
chop_mono_first_thin
                [| $H, $G |- [](A --> B), $E; $H, C ^ B, $G |- $E |]
                ==> $H, C ^ A, $G |- $E
chop_mono_thin  [| $H, $G |- $E, [](A --> B); $H, C ^ B, $G |- $E |]
                ==> $H, C ^ A, $G |- $E
chop_right_assoc
                 |- D ^ (A ^ B) ^ C <-> D ^ A ^ B ^ C
conj_chop       [| RT x; $I, $H, $G |- [](A --> len = x), $E;
                   $I, $H, $G |- [](A' --> len = x), $E;
                   (A & A') ^ (B & B'), $I, $H, $G |- $E |]
                ==> $I, A ^ B, $H, A' ^ B', $G |- $E
diaL            [| $G|L>$G'; $H|L>$H'; $E|R>$E'; $H', P, $G' |- $E' |]
                ==> $H, <>P, $G |- $E
diaR            $E |- $F, <>P, $G, P ==> $E |- $F, <>P, $G
diaR_thin       $E |- $F, P, $G ==> $E |- $F, <>P, $G
dia_true         |- <>A <-> <>(A ^ True)
disj_chop        |- (A | B) ^ C <-> A ^ C | B ^ C
disj_chopL      [| $H, A ^ C, $G |- $E; $H, B ^ C, $G |- $E |]
                ==> $H, (A | B) ^ C, $G |- $E
disj_chopR      $G |- $E, A ^ C, B ^ C, $F ==> $G |- $E, (A | B) ^ C, $F
ex_dia           |- (EX x. <>A(x)) <-> <>(EX x. A(x))
exists_chop     $H, EX x. A(x) ^ B, $G, (EX x. A(x)) ^ B |- $E
                ==> $H, (EX x. A(x)) ^ B, $G |- $E
exists_chop_equiv
                 |- (EX x. A(x)) ^ B <-> (EX x. A(x) ^ B)
exists_chop_thin
                $H, EX x. A(x) ^ B, $G |- $E ==> $H, (EX x. A(x)) ^ B, $G |- $E
exists_len       |- EX x. len = x
exists_nonneg_chop_equiv
                 |- (EX x. x >= 0 & A(x)) ^ B <-> (EX x. x >= 0 & A(x) ^ B)
exists_nonneg_len_chop_equiv
                 |- A ^ B <-> (EX x. x >= 0 & len = x & A) ^ B
exists_nonneg_len_equiv
                 |- A <-> (EX x. x >= 0 & len = x & A)
false_chop      False ^ A |-
intro_len       $H, A ^ (EX x. len = x & B), $G, A ^ B |- $E
                ==> $H, A ^ B, $G |- $E
intro_len_intro_nonneg
                $H, A ^ (EX x. len = x & x >= 0 & B) ^ C, $G |- $E
                ==> $H, A ^ B ^ C, $G |- $E
intro_len_nonneg
                $H, A ^ (EX x. len = x & x >= 0 & B), $G |- $E
                ==> $H, A ^ B, $G |- $E
intro_len_thin  $H, A ^ (EX x. len = x & B), $G |- $E ==> $H, A ^ B, $G |- $E
ldiaR           $E |- $F, True ^ A, $G ==> $E |- $F, <>A, $G
```

```
left_assoc       |- A ^ B ^ C <-> (A ^ B) ^ C
left_assoc_chop
                 |- (A ^ B ^ C) ^ D <-> ((A ^ B) ^ C) ^ D
lemma_chop      [|  |- D; $H, (D & A) ^ B, $G |- $E |] ==> $H, A ^ B, $G |- $E
len_add         [| RT x; RT y |] ==>  |- len = x ^ len = y --> len = x + y
len_add_rl      [| A |- len = x; B |- len = y; RT x; RT y |]
                ==> A ^ B |- len = x + y
len_chop        (!!x. RT x ==> $H, (len = x & A) ^ B, $G, x >= 0 |- $E)
                ==> $H, A ^ B, $G |- $E
len_intro       $H, (EX x. len = x & A) ^ B, $G, A ^ B |- $E
                ==> $H, A ^ B, $G |- $E
len_intro_nonneg
                $H, (EX x. len = x & x >= 0 & A) ^ B, $G |- $E
                ==> $H, A ^ B, $G |- $E
len_intro_thin  $H, (EX x. len = x & A) ^ B, $G |- $E ==> $H, A ^ B, $G |- $E
mono_RT_chop    [| RT R;  |- A --> R; $H, A ^ B, $G, R |- $E |]
                ==> $H, A ^ B, $G |- $E
mono_chop       [| $H, $G |- $E, [](A --> B); $H, B ^ C, $G, A ^ C |- $E |]
                ==> $H, A ^ C, $G |- $E
mono_chop_bang  [|  |- A --> B; $H, B ^ C, $G, A ^ C |- $E |]
                ==> $H, A ^ C, $G |- $E
mono_chop_bang_thin
                [|  |- A --> B; $H, B ^ C, $G |- $E |] ==> $H, A ^ C, $G |- $E
mono_chop_first_thin
                [| $H, $G |- [](A --> B), $E; $H, B ^ C, $G |- $E |]
                ==> $H, A ^ C, $G |- $E
mono_chop_thin  [| $H, $G |- $E, [](A --> B); $H, B ^ C, $G |- $E |]
                ==> $H, A ^ C, $G |- $E
monotone        [| $H, $G |- $E, [](A --> A'), $F;
                   $H, $G |- $E, [](B --> B'), $F |]
                ==> $H, A ^ B, $G |- $E, A' ^ B', $F
monotone_bang   [|  |- A --> A';  |- B --> B' |]
                ==> $H, A ^ B, $G |- $E, A' ^ B', $F
monotone_first  [| $H, $G |- [](A --> A'), $E, $F;
                   $H, $G |- [](B --> B'), $E, $F |]
                ==> $H, A ^ B, $G |- $E, A' ^ B', $F
rdiaR           $E |- $F, A ^ True, $G ==> $E |- $F, <>A, $G
right_assoc      |- (A ^ B) ^ C <-> A ^ B ^ C
right_assoc_chop
                 |- ((A ^ B) ^ C) ^ D <-> (A ^ B ^ C) ^ D
true_dia         |- <>A <-> <>(True ^ A)
```

C.5 TDC

```
Assertion        |- Oo A <-> Oo B ==> $G |- $E, dur(A) = dur(B), $F
DCA6_bang        |- Oo A <-> Oo B ==> $G |- $E, dur(A) = dur(B), $F
DCA6_unbox      [| $G|L>$G'; $E|R>$E'; $F|R>$F';
                   $G' |- $E', Oo A <-> Oo B, $F' |]
                ==> $G |- $E, dur(A) = dur(B), $F
P_lift           |- Oo A <-> Oo B ==>  |- `A` <-> `B`
```

```
chop_dur         (!!x. RT x ==> $H, A ^ (dur(P) = x & B), $G, x >= 0 |- $E)
                 ==> $H, A ^ B, $G |- $E
chop_dur_chop    (!!x. RT x ==> $H, A ^ (dur(P) = x & B) ^ C, $G, x >= 0 |- $E)
                 ==> $H, A ^ B ^ C, $G |- $E
dur_add_rl       [| A |- dur(P) = x; B |- dur(P) = y; RT x; RT y |]
                 ==> A ^ B |- dur(P) = x + y
dur_chop         (!!x. RT x ==> $H, (dur(P) = x & A) ^ B, $G, x >= 0 |- $E)
                 ==> $H, A ^ B, $G |- $E
dur_intro        $H, (EX x. dur(P) = x & x >= 0 & A) ^ B, $G |- $E
                 ==> $H, A ^ B, $G |- $E
exists_dur        |- EX x. dur(P) = x
intro_dur        $H, A ^ (EX x. dur(P) = x & x >= 0 & B), $G |- $E
                 ==> $H, A ^ B, $G |- $E
intro_dur_intro
                 $H, A ^ (EX x. dur(P) = x & x >= 0 & B) ^ C, $G |- $E
                 ==> $H, A ^ B ^ C, $G |- $E
```

C.6 TDCOracle

```
AND_OR_distribL
                  |- `P AND (Q OR R)` <-> `P AND Q OR P AND R`
AND_OR_distribR
                  |- `(P OR Q) AND R` <-> `P AND R OR Q AND R`
AND_commute       |- `P AND Q` <-> `Q AND P`
AND_left_commute
                  |- `P AND Q AND R` <-> `Q AND P AND R`
DC1               |- `` | True ^ `P` | True ^ `NOT P`
DC10              |- `P` & `Q` --> `P AND Q`
DC11              |- `P AND Q` --> `P`
DC17             [| RT x; RT y |]
                 ==>  |- x > 0 & y > 0 -->
                          len = x + y & `P` --> (len = x & `P`) ^ (len = y & `P`)
DC18              |- `P` ^ True ^ `NOT P` --> `P` ^ `NOT P` ^ True
DC19              |- `` |
                     True ^ `NOT P AND NOT Q` |
                     True ^ `P` |
                     True ^ `NOT P AND NOT Q` ^ `P` |
                     True ^ `Q` ^ `P` | True ^ `Q`
DC1b              |- `` | True ^ `P` | True ^ `NOT P`
DC2               |- `` | `P` ^ True | `NOT P` ^ True
DC3               |- dur(P) + dur(NOT P) = len
DC4               |- dur(P) <= len
DC5               |- dur(P) >= dur(P AND Q)
IMPLIES_AND       |- `P AND Q IMPLIES R` <-> `P IMPLIES Q IMPLIES R`
IMPLIES_AND_distrib
                  |- `P IMPLIES Q AND R` <-> `(P IMPLIES Q) AND (P IMPLIES R)`
IMPLIES_OR        |- `P OR Q IMPLIES R` <-> `(P IMPLIES R) AND (Q IMPLIES R)`
IMPLIES_OR1       |- `(P IMPLIES Q) OR R` <-> `P IMPLIES Q OR R`
IMPLIES_OR2       |- `Q OR (P IMPLIES R)` <-> `P IMPLIES Q OR R`
IR1              [|  |- H(``); !!X.  |- H(X) --> H(X | X ^ `A` | X ^ `NOT A`) |]
```

```
                ==>  |- H(True)
IR2             [|  |- H(‘‘); !!X.  |- H(X) --> H(X | ‘A‘ ^ X | ‘NOT A‘ ^ X) |]
                ==>  |- H(True)
OR_AND_distribL
                 |- ‘P OR Q AND R‘ <-> ‘(P OR Q) AND (P OR R)‘
OR_AND_distribR
                 |- ‘P AND Q OR R‘ <-> ‘(P OR R) AND (Q OR R)‘
OR_commute       |- ‘P OR Q‘ <-> ‘Q OR P‘
OR_left_commute
                 |- ‘P OR Q OR R‘ <-> ‘Q OR P OR R‘
OR_true          |- ‘P OR Q‘ ^ True <-> ‘P‘ ^ True | ‘Q‘ ^ True
P_always         |- ‘P‘ --> [](‘‘ | ‘P‘)
P_boxL          $H, ‘P‘, $G |- $E ==> $H, []‘P‘, $G |- $E
P_chop           |- ‘P‘ ^ ‘P‘ <-> ‘P‘
P_chop_A         |- ‘P‘ ^ ‘P‘ ^ A <-> ‘P‘ ^ A
P_conj           |- ‘P‘ & ‘Q‘ <-> ‘P AND Q‘
P_conjL         $I, ‘P AND Q‘, $H, $G |- $E ==> $I, ‘P‘, $H, ‘Q‘, $G |- $E
P_mono_test      |- ‘P‘ & ‘Q‘ & []‘R‘ --> ‘P AND Q‘
P_monotone      [| $H|L>$H’; $G|L>$G’; $E|R>$E’; $F|R>$F’;
                   $H’, Oo P, $G’ |- $E’, Oo Q, $F’ |]
                ==> $H, ‘P‘, $G |- $E, ‘Q‘, $F
P_monotone_False
                [| $H|L>$H’; $G|L>$G’; $E|R>$E’; $H’, Oo P, $G’ |- $E’ |]
                ==> $H, ‘P‘, $G |- $E
P_monotone_False_bang
                Oo P |-  ==> $H, ‘P‘, $G |- $E
P_monotone_bang
                 |- Oo P --> Oo Q ==> $H, ‘P‘, $G |- $E, ‘Q‘, $F
P_point          |- ‘P‘ & ‘‘ <-> False
RT_prop_test    A ^ no(x) ^ C ^ D ^ E |- no(x)
True_P           |- True ^ ‘P‘ <-> (True ^ ‘NOT P‘ | ‘‘) ^ ‘P‘
True_tt          |- True <-> ‘‘ | ‘tt‘
chain_P          |- ‘Q‘ ^ A & ‘P‘ <-> ‘P AND Q‘ ^ ((‘P‘ | ‘‘) & A)
chain_chain      |- ‘P‘ ^ A & ‘Q‘ ^ B <->
                    ‘P AND Q‘ ^ (A & B) |
                    ‘P AND Q‘ ^ (B & ‘P‘ ^ A) | ‘P AND Q‘ ^ (A & ‘Q‘ ^ B)
chain_point      |- ‘P‘ ^ A & ‘‘ <-> False
chop_induct_boxed
                [| $G |- []H(‘‘), $D;
                   !!X. $G |- []([]H(X) --> H(X | X ^ ‘A‘ | X ^ ‘NOT A‘)), $D |]
                ==> $G |- []H(True), $D
chop_point       |- A ^ ‘‘ <-> A
cover_o          |- ‘‘ | (EX o. True ^ ‘P ;= so(o)‘)
de_Morgan_AND    |- ‘NOT (P AND Q)‘ <-> ‘NOT P OR NOT Q‘
de_Morgan_OR     |- ‘NOT (P OR Q)‘ <-> ‘NOT P AND NOT Q‘
dur_implies      |- dur(P IMPLIES Q) = len --> dur(P) <= dur(Q)
ff_False         |- ‘ff‘ <-> False
induct_chop_boxed
                [| $G |- []H(‘‘), $D;
                   !!X. $G |- []([]H(X) --> H(X | ‘A‘ ^ X | ‘NOT A‘ ^ X)), $D |]
                ==> $G |- []H(True), $D
```

```
induct_chop_unboxed
                [| G |- H(``), D;
                   !!X. G |- H(X) --> H(X | `A` ^ X | `NOT A` ^ X), D |]
                ==> G |- H(True), D
not_iff          |- `NOT (P IFF Q)` <-> `P IFF NOT Q`
o_chop_induct   [| $G |- H(``), $D;
                   !!X. $G |- H(X) --> H(X | (EX x. X ^ `s := so(x)`)), $D |]
                ==> $G |- H(True), $D
o_cover          |- `` | (EX o. `P := so(o)` ^ True)
o_induct_chop   [| $G |- H(``), $D;
                   !!X. $G |- H(X) --> H(X | (EX x. `s := so(x)` ^ X)), $D |]
                ==> $G |- H(True), $D
p_monotone_bang
                 |- Oo P --> Oo Q ==> $G |- $E, `P` --> `Q`, $F
point_P          |- `` & `P` <-> False
point_chain      |- `` & `P` ^ A <-> False
point_chop       |- `` ^ A <-> A
true_OR          |- True ^ `P OR Q` <-> True ^ `P` | True ^ `Q`
```

C.7 SREAL

```
real_chop_cover
                 |- EX r. `` | True ^ `R := sr(r)`
real_cover_chop
                 |- EX r. `` | `R := sr(r)` ^ True
```

C.8 EDC

```
b_chop          (!!x. RT x ==> $H, (b. s = x & A) ^ B, $G |- $E)
                ==> $H, A ^ B, $G |- $E
b_intro          |- A <-> (EX x. b. s = x & A)
chop_b          (!!x. RT x ==> $H, A ^ (b. s = x & B), $G |- $E)
                ==> $H, A ^ B, $G |- $E
chop_b_chop     (!!x. RT x ==> $H, A ^ (b. s = x & B) ^ C, $G |- $E)
                ==> $H, A ^ B ^ C, $G |- $E
chop_e          (!!x. RT x ==> $H, A ^ (e. s = x & B), $G |- $E)
                ==> $H, A ^ B, $G |- $E
chop_e_chop     (!!x. RT x ==> $H, A ^ (e. s = x & B) ^ C, $G |- $E)
                ==> $H, A ^ B ^ C, $G |- $E
e_chop          (!!x. RT x ==> $H, (e. s = x & A) ^ B, $G |- $E)
                ==> $H, A ^ B, $G |- $E
e_intro          |- A <-> (EX x. e. s = x & A)
exists_b         |- EX x. b. s = x
exists_e         |- EX x. e. s = x
```

C.9 FB

```
box_fb           |- (A ---> B) <-> [](A ---> B)
```

```
box_fbL        $H, [](A ---> B), $G |- $E ==> $H, A ---> B, $G |- $E
box_lt          |- (A -= t -> B) <-> [](A -= t -> B)
box_ltL        $H, [](A -= t -> B), $G |- $E ==> $H, A -= t -> B, $G |- $E
box_ut          |- (A -<= t -> B) <-> [](A -<= t -> B)
box_utL        $H, [](A -<= t -> B), $G |- $E ==> $H, A -<= t -> B, $G |- $E
fb_monotone    [|  |- B --> True ^ A;  |- Oo P --> Oo Q |]
               ==>  |- (A ---> `P`) --> B ---> `Q`
fb_neg          |- ~ (A ---> `P`) <-> <>(A ^ `NOT P`)
fb_negL        $H, $G |- <>(A ^ `NOT P`), $E ==> $H, A ---> `P`, $G |- $E
fb_negR        <>(A ^ `NOT P`), $G |- $E, $F ==> $G |- $E, A ---> `P`, $F
fb_neg_trueL   $H, $G |- <>((A ^ `NOT P`) ^ True), $E
               ==> $H, A ---> `P`, $G |- $E
fb_pred         |- (A ---> `P`) <-> ~ <>(A ^ `NOT P`)
lt_bound       RT t
               ==>  |- t > 0 --> (`P` -= t -> `NOT P`) <-> [](`P` --> len <= t)
lt_continue    RT t
               ==>  |- t > 0 -->
                       (`P` -= t -> `Q`) -->
                       (ALL r. [](((`P` & len = r) ^ len > 0 -->
                                 r < t | len = r ^ `Q` ^ True))
lt_negL        $H, $G |- <>((A & len = t) ^ `NOT P`), $E
               ==> $H, A -= t -> `P`, $G |- $E
lt_negR        <>((A & len = t) ^ `NOT P`), $G |- $E, $F
               ==> $G |- $E, A -= t -> `P`, $F
lt_prop        RT t
               ==>  |- t > 0 -->
                       (`P` -= t -> `Q`) <->
                       (ALL r. [](((`P` & len = r) ^ len > 0 -->
                                 r < t |
                                 r = t & len = t ^ `Q` ^ True |
                                 r > t &
                                 len = t ^
                                 (`P AND Q` & len = r - t) ^ `Q` ^ True))
lt_propL       [| RT t; $H, $G |- t > 0, $E;
                  $H, ALL r. [](((`P` & len = r) ^ len > 0 -->
                                r < t |
                                r = t & len = t ^ `Q` ^ True |
                                r > t &
                                len = t ^
                                (`P AND Q` & len = r - t) ^ `Q` ^ True),
                  $G, `P` -= t -> `Q`
                  |- $E |]
               ==> $H, `P` -= t -> `Q`, $G |- $E
lt_prop_lem1   RT t
               ==>  |- t > 0 & (`P` -= t -> `Q`) -->
                       [](((`P` & len = t) ^ len > 0 --> len = t ^ `Q` ^ True)
lt_prop_lem2   [| RT t; RT r |]
               ==>  |- r > t & t > 0 & (`P` -= t -> `Q`) -->
                       [](((`P` & len = r) ^ len > 0 -->
                          len = t ^ (`P AND Q` & len = r - t) ^ `Q` ^ True)
propagation    RT t
```

```
                ==>  |- t > 0 -->
                        (`P` -= t -> `Q`) -->
                        [](`P` --> len <= t | len = t ^ `P AND Q`)
stability       RT t
                ==>  |- (`NOT P` ^ `P AND Q` -<= t -> `P`) -->
                        [](`NOT P` ^ (`P` ^ True & `Q`) & len <= t -->
                           `NOT P` ^ `P`)
stable          RT t
                ==>  |- (`NOT P` ^ `P` -<= t -> `P`) <->
                        [](`NOT P` ^ `P` ^ `NOT P` --> len > t)
stable2         RT t
                ==>  |- (`NOT P` ^ `P` -<= t -> `P`) <->
                        [](len > 0 & len <= t -->
                           `P` |
                           `NOT P` |
                           `NOT P` ^ `P` | `P` ^ `NOT P` | `P` ^ `NOT P` ^ `P`)
stable2_LR      RT t
                ==>  |- (`NOT P` ^ `P` -<= t -> `P`) -->
                        [](len > 0 & len <= t -->
                           `P` |
                           `NOT P` |
                           `NOT P` ^ `P` | `P` ^ `NOT P` | `P` ^ `NOT P` ^ `P`)
stable2_RL      RT t
                ==>  |- [](len > 0 & len <= t -->
                           `P` |
                           `NOT P` |
                           `NOT P` ^ `P` |
                           `P` ^ `NOT P` | `P` ^ `NOT P` ^ `P`) -->
                        `NOT P` ^ `P` -<= t -> `P`
stable3         RT t
                ==>  |- (`NOT P` ^ `P` -<= t -> `P`) -->
                        (ALL r. [](`NOT P` ^ (len = r & `P`) ^ `NOT P` -->
                                   r >= t))
stable_LR       RT t
                ==>  |- (`NOT P` ^ `P` -<= t -> `P`) -->
                        [](`NOT P` ^ `P` ^ `NOT P` --> len > t)
stable_RL       RT t
                ==>  |- [](`NOT Q` ^ `Q` ^ `NOT Q` --> len > t) -->
                        `NOT Q` ^ `Q` -<= t -> `Q`
ut_negL         $H, $G |- <>((A & len <= t) ^ `NOT P`), $E
                ==> $H, A -<= t -> `P`, $G |- $E
ut_negR         <>((A & len <= t) ^ `NOT P`), $G |- $E, $F
                ==> $G |- $E, A -<= t -> `P`, $F
```

Appendix D

Steam Boiler Theorems

This appendix contains what is basically a dump of Isabelles internal theorem data base for the Steam Boiler theories. The dump provides evidence that the listed theorems have in fact been derived using Isabelle.

The following sections lists theorems of particular theories. Lists of theorems are sorted alphabetically by name. Only derived theorems are listed to avoid redundancy and to save space. Axioms are declared in the Isabelle theories listed in Appendix B.

D.1 SB_TLT

```
TLTLem11         |- I(a, pRd, pCmd, vCmd, ladj, hadj) --> ~ pRd
TLTLem21         |- R(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj',
                        hadj') &
                    pRd -->
                    pRd'
TLTLem41         |- R(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj',
                        hadj') &
                    ~ pRd & pRd' -->
                    ~ vCmd
TLTLem42         |- (I(a, pRd, vCmd, pCmd, ladj, hadj) --> pRd --> ~ vCmd) &
                    (R(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj',
                         hadj') &
                     (pRd --> ~ vCmd) -->
                     pRd' --> ~ vCmd')
TLTLem43         |- R(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj',
                        hadj') -->
                    vCmd & ~ pRd --> ~ pRd'
TLTLem5          |- R(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj',
                        hadj') &
                    pRd' & ~ subintv(ladj', hadj', nr(lN), nr(hN)) -->
                    pRd
TLTLem51         |- R(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj',
                        hadj') &
                    pRd' & hadj' > nr(hN) -->
                    pRd
```

```
TLTLem52        |- R(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj',
                      hadj') &
                   pRd' & ladj' < nr(lN) -->
                   pRd
TLTLem55        |- R(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj',
                      hadj') &
                   pRd & higher(ladj', hadj', nr(lN), nr(hN)) & a' > a -->
                   ~ pCmd'
TLTLem56        |- R(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj',
                      hadj') &
                   pRd &
                   ~ vCmd & lower(ladj', hadj', nr(lN), nr(hN)) & a' > a -->
                   pCmd'
TLTLem57        |- R(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj',
                      hadj') &
                   pRd &
                   pCmd & lower(ladj', hadj', nr(lN), nr(hN)) & a' > a -->
                   pCmd'
TLTLem58        |- ladj' <= hadj' &
                   nr(lN) <= nr(hN) &
                   R(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj',
                      hadj') &
                   pRd & ~ vCmd & hadj' < nr(lN) & a' > a -->
                   pCmd'
TLTLem59       ladj' <= hadj', nr(lN) <= nr(hN)
               |- R(a, pRd, vCmd, pCmd, a', pRd', vCmd', pCmd', ladj', hadj') &
                  pRd & pCmd & hadj' < nr(lN) & a' > a -->
                  pCmd'
```

D.2 SB

```
AC2_s           |- `eq_s(n, pRd, vCmd, pCmd, ladj, hadj)` -= nr(tau) ->
                   `NOT eq_s(n, pRd, vCmd, pCmd, ladj, hadj)`
AC3_s           |- `NOT eq_s(n, pRd, vCmd, pCmd, ladj, hadj)` ^
                   `eq_s(n, pRd, vCmd, pCmd, ladj, hadj)` -<= nr(tau) ->
                   `eq_s(n, pRd, vCmd, pCmd, ladj, hadj)`
Commit1        Init |- `` | `NOT ready` ^ True
Commit2        Init |- `ready` ---> `ready`
Commit4         |- Init --> `ready` ---> `d := sr(0)`
Init_pRd        |- Init --> `` | `NOT spRd` ^ True
Lem42           |- `` | `spRd IMPLIES NOT svCmd`
PA1_continue   RT r
               ==>  |- (`spCmd` & len = r) ^ len > 0 -->
                       r < nr(tau) | len = r ^ `p := sr(P)` ^ True
PA2_continue   RT r
               ==>  |- (`NOT spCmd` & len = r) ^ len > 0 -->
                       r < nr(tau) | len = r ^ `p := sr(0)` ^ True
P_s_chop_cover  |- `Q` <->
                   (EX a pRd vCmd pCmd ladj hadj.
                       `Q` ^
```

```
                         `Q AND
                          sa ;= sr(a) AND
                          spRd ;= so(pRd) AND
                          svCmd ;= so(vCmd) AND
                          spCmd ;= so(pCmd) AND
                          sladj ;= sr(ladj) AND shadj ;= sr(hadj)`)
P_s_cover_chop    |- `Q` <->
                     (EX a pRd vCmd pCmd ladj hadj.
                         `Q AND
                          sa ;= sr(a) AND
                          spRd ;= so(pRd) AND
                          svCmd ;= so(vCmd) AND
                          spCmd ;= so(pCmd) AND
                          sladj ;= sr(ladj) AND shadj ;= sr(hadj)` ^
                         `Q`)
Sem2_exp          |- `sa ;= sr(n) AND
                      spRd ;= so(pRd) AND
                      svCmd ;= so(vCmd) AND
                      spCmd ;= so(pCmd) AND
                      sladj ;= sr(ladj) AND shadj ;= sr(hadj)` ^
                     `sa ;= sr(n') AND
                      spRd ;= so(pRd') AND
                      svCmd ;= so(vCmd') AND
                      spCmd ;= so(pCmd') AND
                      sladj ;= sr(ladj') AND shadj ;= sr(hadj')` -->
                     R(nr(n), no(pRd), no(vCmd), no(pCmd), nr(n'), no(pRd'),
                       no(vCmd'), no(pCmd'), nr(ladj'), nr(hadj'))
Sync_s            |- `eq_s(n, pRd, vCmd, pCmd, ladj, hadj)` --->
                     `NOT sa ;= sr(n) OR eq_s(n, pRd, vCmd, pCmd, ladj, hadj)`
VA1_continue     RT r
                 ==>  |- (`svCmd` & len = r) ^ len > 0 -->
                         r < nr(tau) | len = r ^ `d ;= sr(D)` ^ True
VA2_continue     RT r
                 ==>  |- (`NOT svCmd` & len = r) ^ len > 0 -->
                         r < nr(tau) | len = r ^ `d ;= sr(0)` ^ True
a_bound           |- `sa ;= sr(n)` --> len <= nr(tau)
a_chop_a          |- `sa ;= sr(x)` ^ `sa ;= sr(y)` -->
                     nr(y) = nr(x) | nr(y) = nr(x + 1)
a_decrease        |- `sa ;= sr(n)` ^ (`sa ;= sr(n')` & len = nr(tau)) -->
                     nr(n) = nr(n' - 1)
a_right_intro     |- EX n. `` |
                           (`` | True ^ `NOT sa ;= sr(n)`) ^
                           (`sa ;= sr(n)` & len <= nr(tau))
a_right_intro_len
                  |- EX n r.
                        `` |
                        (`` | True ^ `NOT sa ;= sr(n)`) ^
                        (`sa ;= sr(n)` & len = r & r <= nr(tau))
a_right_intro_two_tau
                  |- len > nr(tau + tau) -->
                     (EX n n' n'' l1 l2.
```

```
                        True ^
                        `sa ;= sr(n)` ^
                        (len = l1 & `sa ;= sr(n')`) ^
                        (len = l2 & `sa ;= sr(n'')`) &
                        nr(n') = nr(n + 1) &
                        nr(n'') = nr(n' + 1) & l1 = nr(tau) & l2 <= nr(tau))
a_rl_expand      |- True ^ `NOT sa ;= sr(n)` ^ (`sa ;= sr(n)` & A) <->
                    (`sa ;= sr(n - 1)` & len <= nr(tau) |
                     True ^
                     `NOT sa ;= sr(n - 1)` ^
                     (`sa ;= sr(n - 1)` & len = nr(tau))) ^
                    (`sa ;= sr(n)` & A)
a_rl_expand_prop
                 |- True ^ `NOT sa ;= sr(n)` ^ (`sa ;= sr(n)` & A) <->
                    (EX n' r.
                        (`sa ;= sr(n')` &
                         nr(n') = nr(n - 1) & len = r & r <= nr(tau) |
                         True ^
                         `NOT sa ;= sr(n')` ^
                         (`sa ;= sr(n')` &
                          nr(n') = nr(n - 1) & len = r & r = nr(tau))) ^
                        (`sa ;= sr(n)` & A))
a_stable3       RT x
                ==>  |- `NOT sa ;= sr(n)` ^
                        (len = x & `sa ;= sr(n)`) ^ `NOT sa ;= sr(n)` -->
                        x >= nr(tau)
a_then_s         |- `sa ;= sr(n)` -->
                    (EX pRd vCmd pCmd ladj hadj.
                        `eq_s(n, pRd, vCmd, pCmd, ladj, hadj)`)
b_prop          RT x ==>  |- (b. s = x & A) ^ B <-> b. s = x & A ^ B
cont_dec        [| CNT cs; RT vb; RT ve |]
                ==>  |- (`cs >>= sr(x)` & e. cs = ve) ^
                        (`cs <<= sr(x)` & b. cs = vb) -->
                        vb = nr(x) & ve = nr(x)
cont_inc        [| CNT cs; RT vb; RT ve |]
                ==>  |- (`cs <<= sr(x)` & e. cs = ve) ^
                        (`cs >>= sr(x)` & b. cs = vb) -->
                        vb = nr(x) & ve = nr(x)
e_prop          RT x ==>  |- A ^ (e. s = x & B) <-> e. s = x & A ^ B
ready_pRd        |- `ready` <-> `spRd`
s_bound          |- `eq_s(n, pRd, vCmd, pCmd, ladj, hadj)` --> len <= nr(tau)
s_chop_cover     |- `` |
                    (EX a pRd vCmd pCmd ladj hadj.
                        True ^
                        `sa ;= sr(a) AND
                         spRd ;= so(pRd) AND
                         svCmd ;= so(vCmd) AND
                         spCmd ;= so(pCmd) AND
                         sladj ;= sr(ladj) AND shadj ;= sr(hadj)`)
s_cover_chop     |- `` |
                    (EX a pRd vCmd pCmd ladj hadj.
```

```
                        `sa ;= sr(a) AND
                         spRd ;= so(pRd) AND
                         svCmd ;= so(vCmd) AND
                         spCmd ;= so(pCmd) AND
                         sladj ;= sr(ladj) AND shadj ;= sr(hadj)` ^
                        True)
s_lem1          RT r
                ==>  |- `NOT eq_s(n, pRd, vCmd, pCmd, ladj, hadj)` ^
                        (len = r & `eq_s(n, pRd, vCmd, pCmd, ladj, hadj)`) ^
                        `NOT eq_s(n, pRd, vCmd, pCmd, ladj, hadj)` -->
                        r = nr(tau)
s_lem2           |- `eq_s(n, pRd, vCmd, pCmd, ladj, hadj)` ^
                    `eq_s(n', pRd', vCmd', pCmd', ladj', hadj')` &
                    ~ eq_ss(n, pRd, vCmd, pCmd, ladj, hadj, n', pRd', vCmd',
                            pCmd', ladj', hadj') -->
                    nr(n') = nr(n + 1)
s_right_intro_two_tau
                 |- len > nr(tau + tau) -->
                    (EX n pRd vCmd pCmd ladj hadj n' pRd' vCmd' pCmd' ladj'
                        hadj' n'' pRd'' vCmd'' pCmd'' ladj'' hadj'' l1 l2.
                        True ^
                        `eq_s(n, pRd, vCmd, pCmd, ladj, hadj)` ^
                        (len = l1 &
                         `eq_s(n', pRd', vCmd', pCmd', ladj', hadj')`) ^
                        (len = l2 &
                         `eq_s(n'', pRd'', vCmd'', pCmd'', ladj'', hadj'')`) &
                        R(nr(n), no(pRd), no(vCmd), no(pCmd), nr(n'), no(pRd'),
                          no(vCmd'), no(pCmd'), nr(ladj'), nr(hadj')) &
                        R(nr(n'), no(pRd'), no(vCmd'), no(pCmd'), nr(n''),
                          no(pRd''), no(vCmd''), no(pCmd''), nr(ladj''),
                          nr(hadj'')) &
                        nr(n') = nr(n + 1) &
                        nr(n'') = nr(n' + 1) & l1 = nr(tau) & l2 <= nr(tau))
```

D.3 SB_COMMIT5

```
Commit5          |- Init --> `ready` ---> `smember(q, sr(lM), sr(hM))`
Lem51a           |- Init --> `NOT spRd` --->
                    `NOT spRd OR NOT q >> sr(hN + epsilon + epsilon)`
Lem51b           |- Init --> `NOT spRd` --->
                    `NOT spRd OR NOT q << sr(lN - (epsilon + epsilon))`
Lem51c           |- e. q >= nr(hN + epsilon + epsilon) & `NOT spRd` --->
                    `NOT spRd`
Lem51d           |- e. q <= nr(lN - (epsilon + epsilon)) & `NOT spRd` --->
                    `NOT spRd`
Lem53            |- Init --> `spRd` ---> `q <<= sr(hM)`
Lem53a           |- ~ True ^
                      `NOT spRd` ^
                      `spRd AND q <<= sr(hN + epsilon + epsilon)` ^
                      `spRd` ^ `NOT q <<= sr(hM)` ^ True
```

```
Lem53b          RT bq
                ==>  |- ~ (True ^
                           (b. q = bq &
                            `spRd AND NOT q >> sr(hM)` ^
                            `spRd` ^ `NOT q <<= sr(hM)` ^ True) &
                           bq <= nr(hN + epsilon + epsilon))
Lem53c          RT bq
                ==>  |- ~ (True ^
                           (b. q = bq &
                            `spRd AND NOT q >> sr(hM)` ^
                            `NOT q <<= sr(hM)` ^ True) &
                           bq <= nr(hN + epsilon + epsilon))
Lem53d          [| RT bq; RT eq |]
                ==>  |- ~ (True ^
                           (b. q = bq & e. q = eq & `spRd`) ^
                           (b. q = eq & `NOT q <<= sr(hM)`) ^ True &
                           bq <= nr(hN + epsilon + epsilon) & eq = nr(hM))
Lem53e          [| RT r; RT eq |]
                ==>  |- ~ (True ^
                           (len = r &
                            `spRd AND q >> sr(hN + epsilon + epsilon)`) ^
                           (b. q = eq & `NOT q <<= sr(hM)`) ^ True &
                           eq = nr(hM) & r > nr(tau + tau))
Lem54            |- Init --> `spRd` ---> `q >>= sr(lM)`
Lem54a           |- ~ True ^
                      `NOT spRd` ^
                      `spRd AND
                       d ;= sr(0) AND q >>= sr(lN - epsilon - epsilon)` ^
                      `spRd AND d ;= sr(0)` ^
                      `d ;= sr(0) AND NOT q >>= sr(lM)` ^ True
Lem54b          RT bq
                ==>  |- ~ (True ^
                           (b. q = bq &
                            `spRd AND d ;= sr(0) AND NOT q << sr(lM)` ^
                            `spRd AND d ;= sr(0)` ^
                            `d ;= sr(0) AND NOT q >>= sr(lM)` ^ True) &
                           bq >= nr(lN - epsilon - epsilon))
Lem54c          RT bq
                ==>  |- ~ (True ^
                           (b. q = bq &
                            `spRd AND d ;= sr(0) AND NOT q << sr(lM)` ^
                            `d ;= sr(0) AND NOT q >>= sr(lM)` ^ True) &
                           bq >= nr(lN - epsilon - epsilon))
Lem54d          [| RT bq; RT eq |]
                ==>  |- ~ (True ^
                           (b. q = bq & e. q = eq & `spRd AND d ;= sr(0)`) ^
                           (b. q = eq & `d ;= sr(0) AND NOT q >>= sr(lM)`) ^
                           True &
                           bq >= nr(lN - epsilon - epsilon) & eq = nr(lM))
Lem54e          [| RT r; RT eq |]
                ==>  |- ~ (True ^
```

```
                          (len = r &
                           `spRd AND q << sr(lN - epsilon - epsilon)`) ^
                          (b. q = eq & `d := sr(0) AND NOT q >>= sr(lM)`) ^
                          True &
                          eq = nr(lM) & r > nr(tau + tau))
Lem55            |- len > nr(tau + tau) &
                    `spRd AND q >> sr(hN + epsilon + epsilon)` --->
                    `p := sr(0)`
Lem55_lem        |- `spRd AND eq_s(n, pRd, vCmd, pCmd, ladj, hadj)` ^
                    `spRd AND
                     q >> sr(hN + epsilon + epsilon) AND
                     eq_s(n', pRd', vCmd', pCmd', ladj', hadj')` &
                    R(nr(n), no(pRd), no(vCmd), no(pCmd), nr(n'), no(pRd'),
                      no(vCmd'), no(pCmd'), nr(ladj'), nr(hadj')) &
                    nr(n') = nr(n + 1) -->
                    ~ no(pCmd')
Lem55a           |- ~ (True ^
                       `spRd AND eq_s(n, pRd, vCmd, pCmd, ladj, hadj)` ^
                       (len = nr(l1) &
                        `spRd AND
                         q >> sr(hN + epsilon + epsilon) AND
                         eq_s(n', pRd', vCmd', pCmd', ladj', hadj')`) ^
                       (len = nr(l2) &
                        `spRd AND
                         q >> sr(hN + epsilon + epsilon) AND
                         eq_s(n'', pRd'', vCmd'', pCmd'', ladj'', hadj'')`) ^
                       `NOT p := sr(0)` ^ True &
                       R(nr(n), no(pRd), no(vCmd), no(pCmd), nr(n'), no(pRd'),
                         no(vCmd'), no(pCmd'), nr(ladj'), nr(hadj')) &
                       R(nr(n'), no(pRd'), no(vCmd'), no(pCmd'), nr(n''),
                         no(pRd''), no(vCmd''), no(pCmd''), nr(ladj''),
                         nr(hadj'')) &
                       nr(n') = nr(n + 1) &
                       nr(n'') = nr(n' + 1) &
                       nr(l1) = nr(tau) & nr(l2) <= nr(tau))
Lem55b           |- ~ (True ^
                       (len = nr(l1) &
                        `NOT spCmd AND
                         spRd AND eq_s(n', pRd', vCmd', pCmd', ladj', hadj')`) ^
                       (len = nr(l2) &
                        `spRd AND
                         q >> sr(hN + epsilon + epsilon) AND
                         eq_s(n'', pRd'', vCmd'', pCmd'', ladj'', hadj'')`) ^
                       `NOT p := sr(0)` ^ True &
                       R(nr(n'), no(pRd'), no(vCmd'), no(pCmd'), nr(n''),
                         no(pRd''), no(vCmd''), no(pCmd''), nr(ladj''),
                         nr(hadj'')) &
                       nr(n'') = nr(n' + 1) &
                       nr(l1) = nr(tau) & nr(l2) <= nr(tau))
Lem55c          [| RT l1; RT l2 |]
                ==>  |- ~ (True ^
```

```
                    (len = l1 & `NOT spCmd`) ^
                    (len = l2 & `NOT spCmd`) ^ `NOT p ;= sr(0)` ^ True &
                    l1 = nr(tau) & l2 <= nr(tau))
Lem58          |- len > nr(tau + tau) &
                  `spRd AND q << sr(lN - (epsilon + epsilon))` --->
                  `p ;= sr(P)`
Lem58_lem      |- `spRd AND eq_s(n, pRd, vCmd, pCmd, ladj, hadj)` ^
                  `spRd AND
                   q << sr(lN - (epsilon + epsilon)) AND
                   eq_s(n', pRd', vCmd', pCmd', ladj', hadj')` &
                  R(nr(n), no(pRd), no(vCmd), no(pCmd), nr(n'), no(pRd'),
                    no(vCmd'), no(pCmd'), nr(ladj'), nr(hadj')) &
                  nr(n') = nr(n + 1) -->
                  no(pCmd')
Lem58a         |- ~ (True ^
                     `spRd AND eq_s(n, pRd, vCmd, pCmd, ladj, hadj)` ^
                     (len = nr(l1) &
                      `spRd AND
                       q << sr(lN - (epsilon + epsilon)) AND
                       eq_s(n', pRd', vCmd', pCmd', ladj', hadj')`) ^
                     (len = nr(l2) &
                      `spRd AND
                       q << sr(lN - (epsilon + epsilon)) AND
                       eq_s(n'', pRd'', vCmd'', pCmd'', ladj'', hadj'')`) ^
                     `NOT p ;= sr(P)` ^ True &
                     R(nr(n), no(pRd), no(vCmd), no(pCmd), nr(n'), no(pRd'),
                       no(vCmd'), no(pCmd'), nr(ladj'), nr(hadj')) &
                     R(nr(n'), no(pRd'), no(vCmd'), no(pCmd'), nr(n''),
                       no(pRd''), no(vCmd''), no(pCmd''), nr(ladj''),
                       nr(hadj'')) &
                     nr(n') = nr(n + 1) &
                     nr(n'') = nr(n' + 1) &
                     nr(l1) = nr(tau) & nr(l2) <= nr(tau))
Lem58b         |- ~ (True ^
                     (len = nr(l1) &
                      `spCmd AND
                       spRd AND eq_s(n', pRd', vCmd', pCmd', ladj', hadj')`) ^
                     (len = nr(l2) &
                      `spRd AND
                       q << sr(lN - (epsilon + epsilon)) AND
                       eq_s(n'', pRd'', vCmd'', pCmd'', ladj'', hadj'')`) ^
                     `NOT p ;= sr(P)` ^ True &
                     R(nr(n'), no(pRd'), no(vCmd'), no(pCmd'), nr(n''),
                       no(pRd''), no(vCmd''), no(pCmd''), nr(ladj''),
                       nr(hadj'')) &
                     nr(n'') = nr(n' + 1) &
                     nr(l1) = nr(tau) & nr(l2) <= nr(tau))
Lem58c        [| RT l1; RT l2 |]
              ==>  |- ~ (True ^
                         (len = l1 & `spCmd`) ^
                         (len = l2 & `spCmd`) ^ `NOT p ;= sr(P)` ^ True &
```

```
                            l1 = nr(tau) & l2 <= nr(tau))
WLS1_s            |- e. q = nr(q0) & `eq_s(n, pRd, vCmd, pCmd, ladj, hadj)` --->
                     `eq_s(n, pRd, vCmd, pCmd, ladj, hadj) OR
                      smember(sr(q0), sladj, shadj)`
b_prop_rlL       [| RT x; $H, A ^ B, $G |- b. s = x, $E;
                    $H, (b. s = x & A) ^ B, $G |- $E |]
                 ==> $H, A ^ B, $G |- $E
e_prop_rlL       [| RT x; $H, A ^ B, $G |- e. s = x, $E;
                    $H, A ^ (e. s = x & B), $G |- $E |]
                 ==> $H, A ^ B, $G |- $E
init_pRd          |- Init --> `` | `NOT spRd` ^ True
intg_d_range      |- nr(0) <= intg(d) & intg(d) <= nr(D) * len
intg_p_range      |- nr(0) <= intg(p) & intg(p) <= nr(P) * len
intg_v_range      |- nr(0) <= intg(v) & intg(v) <= nr(W) * len
maxPDWpos         |- max(max(nr(P), nr(D)), nr(W)) > 0
maxPDpos          |- max(nr(P), nr(D)) > 0
max_PDW           |- (nr(P) > nr(D) --> max(nr(P), nr(D)) = nr(P)) &
                     (nr(P) <= nr(D) --> max(nr(P), nr(D)) = nr(D)) &
                     (max(nr(P), nr(D)) > nr(W) -->
                      max(max(nr(P), nr(D)), nr(W)) = max(nr(P), nr(D))) &
                     (max(nr(P), nr(D)) <= nr(W) -->
                      max(max(nr(P), nr(D)), nr(W)) = nr(W))
taumaxPDWpos      |- 2 * nr(tau) * max(max(nr(P), nr(D)), nr(W)) > 0
taumaxPDpos       |- nr(tau) * max(nr(P), nr(D)) > 0
twotaumaxPDpos    |- 2 * nr(tau) * max(nr(P), nr(D)) > 0
```

Appendix E

Soundness Proofs

This appendix provide evidence that soundness proofs of selected rules have been performed. Proofs are done in PVS using the semantic encoding of PC/DC. We show only the encoding of the rules and their status as proved theorems. In PVS a formula is proved `complete` is its proof only rely on axioms and lemmas which are proved complete.

E.1 IL

Encoding

```
il : THEORY

  BEGIN

    dc : LIBRARY "./dc"
    IMPORTING dc@dc_top

    A, B, C : VAR IForm
    Gamma, Delta : VAR IForm

    monotone_chop : THEOREM
        rule(seq(Gamma,[](A=>B) + Delta),
             seq(Gamma,A^C => B^C + Delta))

    chop_monotone : THEOREM
        rule(seq(Gamma,[](A=>B) + Delta),
             seq(Gamma,C^A => C^B + Delta))

    boxR : THEOREM
        rule(seq([]Gamma,A + <>Delta),
             seq([]Gamma,[]A + <>Delta))
  END il
```

Status

```
 Proof summary for theory il
    monotone_chop.........................................proved - complete
    chop_monotone.........................................proved - complete
    boxR..................................................proved - complete
    Theory totals: 3 formulas, 3 attempted, 3 succeeded.
```

E.2 DC

Induction rules encoding

```
induct_rules : THEORY
  BEGIN

     dc : LIBRARY = "./dc"
     IMPORTING
       dc_bool,
       dc_dc,
       dc_quant[bool]
     IMPORTING dc@dc_bool_induct

     A, B, X: VAR IForm
     n: VAR nat
     P: VAR State[bool]
     Gamma,Delta: VAR IForm
     x : VAR bool

     %%% INDUCTION FORMULA PREDICATE %%%

     R: VAR {f:[IForm -> IForm]| subst(f)}

     %%% LEFT INDUCTION %%%

     ind_init_L: LEMMA rule(
                 seq(Gamma,R(point) + Delta),
                 (FORALL X:
                 seq(Gamma, R(X)=>R(X\/pr(P)^X\/pr(NOT P)^X) + Delta)),
                 seq(Gamma, R(faL(n)(P)) + Delta))


     %%% RIGHT INDUCTION %%%
```

```
    ind_init_R: LEMMA rule(
                seq(Gamma,R(point) + Delta),
                (FORALL X:
                seq(Gamma, R(X)=>R(X\/X^pr(P)\/X^pr(NOT P)) + Delta)),
                seq(Gamma, R(faR(n)(P)) + Delta))


    %%% INDUCTION RULES %%%

    induct_chop : LEMMA rule(
                seq(Gamma,R(point) + Delta),
                (FORALL X:
                seq(Gamma, R(X)=> R(X\/pr(P)^X\/pr(NOT P)^X) + Delta)),
                seq(Gamma,R(true) + Delta))

    chop_induct : LEMMA rule(
                seq(Gamma,R(point) + Delta),
                (FORALL X:
                seq(Gamma, R(X)=> R(X\/X^pr(P)\/X^pr(NOT P)) + Delta)),
                seq(Gamma,R(true) + Delta))

  END induct_rules
```

Induction rules status

```
 Proof summary for theory induct_rules
    ind_init_L.........................................proved - complete
    ind_init_R.........................................proved - complete
    induct_chop........................................proved - complete
    chop_induct........................................proved - complete
    Theory totals: 4 formulas, 4 attempted, 4 succeeded.
```

DCA6 encoding

```
dc : THEORY

  BEGIN

    dc : LIBRARY "./dc"
    IMPORTING dc@dc_top

    x : VAR real
    i : VAR Interval
    ix, iy : VAR IReal
    A, B : VAR State[bool]
    F : VAR IForm

    Oo(A)(i) : bool = A(proj_1(i))

    P, Q: VAR State[bool]
```

```
   Gamma, Delta : VAR IForm

   subeq(P, Q)(i) : bool =
      (FORALL (t:Time):  t >= proj_1(i) AND t <= proj_2(i) IMPLIES P(t)=Q(t))

   eq_dur_ax : LEMMA P=Q IMPLIES dur(P)=dur(Q)
   subeq_dur_ax : AXIOM subeq(P, Q)(i) IMPLIES (dur(P)=dur(Q))(i)

   Oo_eq : LEMMA valid(Oo(P) <=> Oo(Q)) IMPLIES (P=Q)
   Oo_subeq : LEMMA ([](Oo(P) <=> Oo(Q)))(i) IMPLIES subeq(P, Q)(i)

   DCA6 : THEOREM valid(Oo(A) <=> Oo(B)) IMPLIES valid(dur(A)=dur(B))

   box_DCA6 : THEOREM
       rule(seq(Gamma, ([](Oo(P) <=> Oo(Q))) + Delta),
            seq(Gamma, (dur(P)=dur(Q)) + Delta))

   DCA6_seq : THEOREM
       rule(seq([]Gamma, (Oo(P) <=> Oo(Q)) + <>Delta),
            seq([]Gamma, (dur(P)=dur(Q)) + <>Delta))

 Oott : THEOREM Oo(tt) = True
 Ooff : THEOREM Oo(ff) = False

 Oonot : THEOREM Oo(NOT A) = NOT(Oo(A))
 Ooand : THEOREM Oo(A /\ B) = (Oo(A) /\ Oo(B))
 Ooor  : THEOREM Oo(A \/ B) = (Oo(A) \/ Oo(B))
 Ooimplies : THEOREM Oo(A => B) = (Oo(A) => Oo(B))
 Ooiff : THEOREM Oo(A <=> B) = (Oo(A) <=> Oo(B))

 END dc
```

DCA6 status

```
Proof summary for theory dc
   eq_dur_ax..............................................proved - complete
   Oo_eq..................................................proved - complete
   Oo_subeq...............................................proved - complete
   DCA6...................................................proved - complete
   box_DCA6...............................................proved - complete
   DCA6_seq...............................................proved - complete
   Oott...................................................proved - complete
   Ooff...................................................proved - complete
   Oonot..................................................proved - complete
   Ooand..................................................proved - complete
   Ooor...................................................proved - complete
   Ooimplies..............................................proved - complete
   Ooiff..................................................proved - complete
   Theory totals: 13 formulas, 13 attempted, 13 succeeded.
```

Appendix F

CD-ROM

The software described in this thesis is available in full on the accompanying CD-ROM. The README file details the contents:

```
This is the README file on the CD-ROM accompanying the PhD thesis:

              Proof Support for Duration Calculus

                     Soeren T. Heilmann

                      January 3, 1999


CD-ROM contents:
----------------
README          This file
thesis.ps       The thesis in Postscript format
DC/             The collection of Isabelle theories encoding DC
                and the Steam Boiler case study
DC/checking     SML code for the Decidable DC decision procedure by
                Sestoft and Skakkebaek
soundness       The soundness proofs of selected DC rules encoded in PVS2.2
svc_bug         Email from Clark Barrett detailing a bug in SVC 1.0

Supporting software
-------------------
Isabelle98      Generic proof tool by Paulson
                Available at:
                  http://www.cl.cam.ac.uk/Research/HVG/Isabelle/
SVC1.0          Decision procedure for real arithmetic by Dill et al.
                Available at:
                  http://agamemnon.stanford.edu/~levitt/vc/index.html
                The patch suggested in the email from Barrett
                (see file svc_bug) must be applied.
```

Bibliography

[1] Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors. *Formal Methods for Industrial Applications*, volume 1165 of *LNCS*. Springer-Verlag, 1996.

[2] Jean-Raymond Abrial, Egon Börger, and Hans Langmaack. The steam boiler case study: Competition of formal program specification and development methods. In Abrial et al. [1].

[3] S. Agerholm and J. Frost. An Isabelle based theorem prover for VDM-SL. In E. L. Gunter and A. Felty, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1275 of *LNCS*. Springer-Verlag, 1997.

[4] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–234, 1994.

[5] E. Asarin, P. Caspi, and O. Maler. A Kleene theorem for timed automata. In G. Winskel, editor, *12th Annual IEEE Symposium on Logic in Computer Science (LICS'97)*, 1997.

[6] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. Validity checking for combinations of theories with equality. In *Proceedings of FMCAD'96*, November 1996.

[7] Robert A. Bull and Krister Segerberg. Basic modal logic. In D. Gabbay and F.Guenthner, editors, *Handbook of Philosophical Logic*, volume II, pages 1–88. D. Reidel Publishing Co., 1983.

[8] Zhou Chaochen and Michael R. Hansen. Duration Calculus: a formal apprach to real-time systems. Draft, September 1997.

[9] Zhou Chaochen, C. A. R. Hoare, and Anders P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1992.

[10] Zhou Chaochen, Zhang Jingzhong, Yang Lu, and Li Xiaoshan. Linear duration invariants. In Langmaack et al. [29], pages 86–109.

[11] Zhou Chaochen, Anders P. Ravn, and Michael R. Hansen. An extended duration calculus for hybrid real-time systems. In R. L. Grossmann, A. Nerode, A. P. Ravn,

and H. Rischel, editors, *Hybrid Systems*, volume 736 of *LNCS*, pages 36–59. Springer-Verlag, 1993.

[12] Jorge Cuéllar and Isolde Wildgruber. The steam-boiler program – a TLT solution. In Abrial et al. [1]. Appendix A.

[13] Li Xuan Dong and Dang Van Hung. Checking Linear Duration invariants by Linear Programming. In Joxan Jaffar and Roland H. C. Yap, editors, *Concurrency and Parallelism, Programming, Networking and security*, volume 1179 of *LNCS*, pages 321–332. Springer-Verlag, 1996.

[14] Bruno Dutertre. On first order interval temporal logic. Technical Report CSD-TR-94-3, Royal Holloway, University of London, February 1995.

[15] Martin Fränzle. *Controller Design form Temporal Logic: Undecidability Need Not Matter*. PhD thesis, Christian-Albrechts Universitä t, Kiel, Germany, June 1997. Bericht Nr. 9710.

[16] Jacob Frost. *Effective Programming*. PhD thesis, Dept. of Information Technology, Technical University of Denmark, 1996. IT-TR:1996-001.

[17] Jean H. Gallier. *Logic for Computer Science*. Wiley, 1986.

[18] M. J. L. Gordon and T. F. Melham. *Introduction to HOL, a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[19] Brent Hailpern. *Verifying Concurrent Processes Using Temporal Logic*, volume 129 of *LNCS*. Springer-Verlag, 1982. PhD. thesis, Stanford University, 1980.

[20] A. G. Hamilton. *Logic for Mathematicians*. Cambridge University Press, 1988. Revised edition.

[21] Michael R. Hansen and Zhou Chaochen. Duration Calculus, logical foundations. *Formal Aspects of Computing*, 9:283–303, 1997.

[22] John R. Harrison. *Theorem Proving with the Real Numbers*. PhD thesis, Churchill College, Cambridge, England, 1996.

[23] John Hawthorn. Natural deduction in normal modal logic. *Notre Dame Journal of Formal Logic*, 31(2):263–273, Spring 1990.

[24] Anne Haxthausen and Xia Yong. A RAISE specification framework and justification assistant for the Duration Calculus. In Michael R. Hansen, editor, *ESSLLI'98 workshop on Duration Calculus: A logical Approach to Real-Time Systems*, pages 51–58, 1998. Workshop part of 10th European Summer School in Logic, Language and Information, Saarbrücken, Germany.

[25] J. He, C. A. R. Hoare, M. Fränzle, M. Müller-Olm, E-R. Olderog, M. Schenke, M.R. Hansen, A. P. Ravn, and H. Rischel. Provably correct systems. In Langmaack et al. [29], pages 288–335.

[26] G. E. Hughes and M. J. Cresswell. *An Introduction to Modal Logic*. Methuen and Co. Ltd., London, 1968.

[27] N. Klarlund and A. Møller. *MONA Version 1.2 User Manual*. BRICS Notes Series NS-98-3, Department of Computer Science, University of Aarhus, June 1998.

[28] Yassine Lakhnech. *Specification and Verification of Hybrid and Real-Time Systems*. PhD thesis, Christian-Albrechts Universitä t, Kiel, Germany, April 1996. Bericht Nr. 9609.

[29] H. Langmaack, W. P. de Roever, and J. Vytopil, editors. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *LNCS*. Springer-Verlag, 1994.

[30] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *Journal of Software Tools for Technology Transfer*, 1(1+2), 1997.

[31] S. Owre, N. Shankar, and J. M. Rushby. *User Guide for the PVS Specification and Verification System*. Computer Science Laboratory, SRI International, Menlo Park CA 94025, USA, March 1993.

[32] Paritosh K. Pandya. DCVALID, a validity checker for Duration Calculus formulas. http://www.tcs.tifr.res.in/ pandya/dcvalid.html.

[33] L. C. Paulson, editor. *Isabelle, A Generic Theorem Prover*, volume 828 of *LNCS*. Springer-Verlag, 1994.

[34] L. C: Paulson. The Isabelle reference manual. In Paulson [33].

[35] L. C: Paulson. Isabelle's object logics. In Paulson [33].

[36] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989.

[37] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of SIGPLAN'88, Conference on Programming Language Design and Implementation*, pages 199–208, 1988.

[38] Dag Prawitz. *Natural Deduction*. Number 3 in Stockholm studies in Philosophy. Almqvist & Wiksells Boktryckeri AB, Stockholm, 1965. Doctoral thesis.

[39] Ole Rasmussen. *Transformational VLSI Design*. PhD thesis, Department of Information Technology, Technical University of Denmark, April 1997.

[40] Anders P. Ravn. Design of embedded real-time computing systems. Technical Report 170, Dept. of Computer Science, Technical University of Denmark, 1995.

[41] Søren T. Heilmann. The Steam Boiler, a case study in proof support. In Michael R. Hansen, editor, *ESSLLI'98 workshop on Duration Calculus: A logical Appraoch to Real-Time Systems*, pages 59–68, August 1998. Workshop part of 10th European Summer School in Logic, Language and Information, Saarbrücken, Germany.

[42] Hans Rischel, Jorge Cuéllar, Simon Mørk, Anders P. Ravn, and Isolde Wildgruber. Development of safety-critical real-time systems. In M. Bartosek, J. Staudek, and J. Wiedermann, editors, *Proceedings of SOFSEM'95*, volume 1012 of *LNCS*, pages 206–235. Springer-Verlag, 1995.

[43] Michael Schenke and Anders P. Ravn. Refinement from a control problem to programs. In Abrial et al. [1], pages 403–427.

[44] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[45] Jens Ulrik Skakkebæk. *A Verification Assistant for a Real-Time Logic*. PhD thesis, Dept. of Computer Science, Technical University of Denmark, 1994. Technical Report 150.

[46] Jens Ulrik Skakkebæk and Peter Sestoft. Checking validity of Duration Calculus formulas, 1994.

[47] Richard Stallman. *GNU Emacs Manual*. Free Software Foundation, 675 Massachusetts Ave., Cambridge, MA, USA, 6th edition edition, March 1987.

[48] A. Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, 1951. Previous version published as a technical report by the RAND Corporation, 1948; Prepared for publication by J. C. C. McKinsey.

[49] Jan Vitt and Jozef Hooman. Assertional specification and verification using PVS of the Steam Boiler control system. In Abrial et al. [1], pages 453–472.

[50] Mao Xiaoquang, Xu Qiwen, and Wang Ji. Towards a proof assistant for interval logics. Technical Report 77, UNI/IIST, International Institute for Software Technology, P.O. Box 3058, Macau, 1996.

[51] Chaochen Zhou, Michael R. Hansen, and Peter Sestoft. Decidability and undicidability results of duration calculus. In *STACS'93*, volume 665 of *LNCS*, pages 58–68. Springer-Verlag, 1993.