Reasoning about VDM-SL Proof Obligations in HOL

Sten Agerholm   Kim Sunesen

IFAD, Forskerparken 10
DK-5230 Odense M, Denmark

May 25, 1999

Project Id.:	PROSPER LTR 26241
Deliverable Id.:	D1.4a.V1
Document Id.:	IFAD-PROSPER-DOC-5
Availability:	RESTRICTED

Copyright © 1998 IFAD.

Contents

1 Introduction **1**

2 VDM-SL proof obligations **2**

3 Domain checking **5**
3.1 Strategies 5
3.2 Summary of experiments 7
3.2.1 Unfolding definitions 9
3.2.2 Arithmetic 11
3.2.3 Simplification 13

4 Subtype checking **16**
4.1 Strategies 16
4.2 Summary of experiments 19
4.2.1 Reusing type information 24
4.2.2 Normal form 25
4.2.3 Translational choices 27

5 Implicitly defined functions **32**

6 Termination **34**

7 General properties **36**

8 Conclusions **39**

A Appendix **42**

Chapter 1

Introduction

This document presents experiments in proving the validity of VDM-SL proof obligations using the HOL theorem prover. Proof obligations are logical properties that are generated from VDM-SL specifications in order to ensure their internal consistency (see [2]). We see the ability to handle proof obligation reasoning well as a key requirement of our VDM-SL proof tool. However, it is obviously not possible to automate the proofs of all proof obligations, so we aim to be able to automate as many as possible.

There are two main purposes of the experiments. First, they should improve our understanding of how involved such verifications are, before we start to design and implement the VDM-SL proof tool. Second, they should generate input for the development of the PROSPER Core Proof Engine (CPE). Hence, this report aims to illustrate the easy, and especially the difficult aspects of verifying proof obligations, using a range of different proof techniques and proof support tools provided by the current HOL98/CPE system (Athabasca Release version 1).

The rest of the report is structured as follows. Chapter 2 presents a classification of the 24 proof obligations chosen for the experiments and explains the general ideas of our proof attempts. Chapter 3 presents concrete attempts to prove proof obligations categorized as domain checking, and Chapter 4 discusses the strategies for proving proof obligations categorized as subtype checking. Chapter 5 and Chapter 6 present preliminary attempts to prove satisfiability and termination checking conditions, and the developed proof strategies are exercised on some general conjectures in Chapter 7. The conclusions are presented in Chapter 8.

As a prerequisite the reader is expected to be familiar with the HOL98 theorem prover [8], and in particular with the specification examples given in [5] and [3].

Chapter 2

VDM-SL proof obligations

Proof obligations in VDM-SL are logical conjectures that must hold of a model in order for it to be regarded as internally consistent. Proof obligations can be produced automatically by a proof obligation generator (see [2]), which is currently being developed in PROSPER (Task 1.1). However, the experiments presented in this document concern the proof of obligations rather than their generation.

The 24 proof obligations for the experiments reported here were derived manually from examples in [7] and are documented in the PROSPER reports [5] and [3], which present the VDM-SL and HOL98 versions respectively. The example specifications are relatively small, but employ a variety of different constructs in the declarative subset of VDM-SL that we are interested in, and so are well-suited for the purposes of the present experiments.

Our initial proof attempts indicated that proof obligations may be grouped into classes on the basis of the situation in which they arise. Obligations in each class appear often to require similar proof strategies. We therefore started to classify proof obligations in four different categories:

Domain checking: Proof obligations that are generated due to the use of partial operators. If verified, these ensure that the operators are applied to values in their respective domains.

Subtype checking: Proof obligations that are generated due to the use of subtypes, in particular due to the use of invariants. If verified, these ensure that subtype relations hold.

Satisfiability: Proof obligations that are generated for implicit function definitions. If verified, these ensure that there exists a valid result for any input of such functions.

Termination: Proof obligations that are generated for recursive type and function definitions. If verified, these ensure that the recursive definitions terminate. Note that this category is included for completeness only as the

Example	PO	Classification	Duration (sec.)
Alarm	1	domain checking	0.0
	2	domain checking	0.0
	3	domain checking	0.0
	4	Satisfiability	3.0
Gateway	1	domain checking	0.0
	2	domain checking	0.0
	3	domain checking	0.0
Tracker	1	domain checking	2.7
	2	subtype checking	6.6
	3	domain checking	0.0
	4	domain checking	0.0
	5	domain checking	0.1
	6	subtype checking	12.2
	7	subtype checking	101.3
	8	domain checking	0.0
	9	domain checking	0.2
	10	subtype checking	8.5
	11	domain checking	0.2
	12	subtype checking	528.5
	13	domain checking	0.1
	14	subtype checking	127.1
	15	domain checking	0.0
	16	domain checking	0.0
	17	domain checking	0.0

Table 2.1: Summary of experiments in verifying proof obligations.

initial version of the proof obligation generator will not support termination proof obligations (see [2]). Currently termination is checked using HOL directly (see [1]).

In this report, the main focus is on proof obligations falling into the first two categories, but some preliminary experiments have also been conducted with proof obligations in the other two categories, which we believe are less numerous in practice. Note that we expect this classification to be refined in further experiments as our proof experience increases. For example, proof obligations due to map application and the division operator both fall into the first category, but are likely to require fairly different proof strategies.

Table 2.1 presents a classification of the 24 proof obligations chosen for our experiments and shows the duration of the fastest proof measured in the experiments discussed in the following chapters.

The main focus is on proving the validity of proof obligations categorized as domain checking and subtype checking. In general, it is undecidable whether

or not a given proof obligation is valid. Hence, we cannot hope to invent a fully automated tool to decide all proof obligations. Instead we can look for automated techniques that can decide many proof obligations in practice. In searching for an automated strategy to verify all 24 example proof obligations, we have tried to employ standard decision procedures first to prove the easy ones, and then a heuristic automated strategy to handle a wider class of proof obligations.

In the experiments, all but one of the domain checking proof obligations are solved within seconds by propositional logic decision procedures whereas none of the subtype checking proof obligations are handled by these. To solve the subtype checking proof obligations, we suggest a rather brute-force, fully-automated tactic which involves advanced rewriting-based simplification, deduction and resolution techniques. The tactic solves each of the subtype checking proof obligations within minutes. Some proofs involve subgoals which split into as many as 114 subgoals.

The suggested tactic was developed in an "extend-by-need" fashion. While solving the proof obligations one by one, we extended the current tactic with new techniques and theorems when subgoals were not solved by the current tactic. This somewhat arbitrary and time-consuming process led to a tactic which solves all of the example proof obligations. We made a few experiments with the robustness of the tactic by testing the consequences of some minor changes in the tactic and specifications of the proof obligations.

Even though the main focus is on domain checking and subtype checking, we have also done some initial case studies verifying the termination of user-defined functions and the satisfiability of implicitly defined functions, i.e. functions specified by pre- and post-conditions. Furthermore, we exhibit some initial experiments with the inductive proving of conjectures expressing general desired properties of specifications (termed "validation conjectures" in VDM).

Chapter 3

Domain checking

The first class of proof obligations that we investigate is the class generated from the use of partial operators. These cover built-in partial operators, like map application, head, and division, and user-defined partial functions defined by the use of preconditions.

In the examples [5, 3], the proof obligations 1 to 3 of the alarm examples, 1 to 3 of the gateway example and 1, 3 to 5, 8, 9, 11, 13, 15 to 17 of the tracker example are classified as domain checking proof obligations.

3.1 Strategies

Our main interest is the investigation of classes of proof obligations that in practice are equally hard or easy to solve, that is, solvable by the same techniques.

A quick inspection of the domain checking proof obligations in the examples [5, 3] suggests that some can be solved by decision procedures. Therefore, we have tried out six of HOL98's standard decision procedures implemented by the tactics described below.

`TAUT_TAC` is a tautology checker. Given a goal which is an instance of a valid propositional formula, `TAUT_TAC` proves the goal by performing Boolean case analysis on the variables of the goal. A propositional formula is a term containing only Boolean constants, variables, conditionals and standard logical connectives. In particular, let expressions are not accepted. The instance of a formula is the formula with one or more variables replaced by terms of the same type. Goals with or without universal quantifiers for the variables are accepted. The tactic fails if the goal is not an instance of a propositional formula or if the goal is not valid. The tactic is described in the HOL reference library.

`MESON_TAC` is a decision procedure for first order logic. Given a goal which is a valid first order formula it proves the goal using a PTTP (prolog technology theorem prover) based implementation of model elimination (for details see [9]).

The tactic fails if the goal is not a first order formula or if goal is not a valid formula.

DECIDE_TAC is a decision procedure combining a number of cooperating decision procedures. Given a goal which is a quantifier-free formula constructed from linear natural number arithmetic, propositional logic, and the equational theories of pairs, recursive types, and uninterpreted function symbols the procedure either proves that the formula is valid or fails. Also, formulas that when put in prenex normal form contain only universal quantifiers are accepted, and additionally formulas with subterms from other theories are accepted if the subterms do not affect the validity of the formula. The procedure is described in [6].

SIMP_TAC bool_ss [] is an instantiation to standard logic simplification of the powerful simplifier implemented by SIMP_TAC. Given a goal, it rewrites/simplifies the goal using standard logic simplifications like

```
|- (!t. ~~t = t) /\ (~T = F) /\ (~F = T)
```

and

```
|- !t.
     (T ==> t = t) /\
     (t ==> T = T) /\
     (F ==> t = T) /\
     (t ==> t = T) /\
     (t ==> F = ~t)
```

The simplifier is documented in [13]. Another helpful reference is the description of the simplifier of the Isabelle system given in [11]. The tactic never fails but may not advance the goal.

ARITH_TAC [1] is a partial decision procedure for a subset of Presburger arithmetic on the natural numbers. The procedure is based on two separate (incomplete) methods: one for handling universally quantified formulas and one for existentially quantified formulas. The procedure only accepts formulas without genuine quantifier alternations, that is, formulas which when put in prenex form contain only one kind of quantifiers. The tactic is described in the HOL reference library.

REDUCE_TAC performs arithmetic and Boolean reductions, in bottom-up order, on all suitable redexes of a goal. In particular, it proves any goal, if true, constructed from only numerals and the Boolean constants. The tactic is described in the HOL reference library

Based on the amount of unfolding of definitions done before applying a tactic we define three strategies for applying the decision tactics from above

[1] Defined by CONV_TAC ARITH_CONV.

1. without any unfolding of specification definitions,
2. with unfolding of specification definitions of projections and predicates and elimination of let-expressions, and
3. with unfolding of all specification definitions and elimination of let-expressions.

In HOL98, the strategies for (2) and (3) are

```
fun VDM_PROJ_AND_PRED_DEF_RW (tac:tactic):tactic =
  PURE_RW_TAC (append proj_DEF_rules pred_DEF_rules) THEN
  (CONV_TAC (DEPTH_CONV let_CONV)) THEN tac;

fun VDM_DEF_RW (tac:tactic):tactic =
  PURE_RW_TAC DEF_rules THEN
  (CONV_TAC (DEPTH_CONV let_CONV)) THEN tac;
```

3.2 Summary of experiments

In this section, we summarize the experiments that we performed with the solving of domain checking proof obligations.

For each of the decision procedures above, we applied each of the three unfolding tactics to each of the domain proof obligations.

The file `vdm-domain-tactics.sml` containing the HOL98 tactics used, and the files `alarm-proof-lab.sml`, `gateway-proof-lab.sml`, and `tracker-proof-lab.sml` containing the applications of the tactics are found in [4].

Table 3.1 summarizes the results obtained. For each tactic, we have applied each of the three strategies to all proof obligations and marked the response time in seconds. The grey shaded boxes mark that the strategy failed.

The results divide into two groups; those that solved all but one of the domain proof obligations and those that failed on almost all. The first group consists of `TAUT_TAC`, `MESON_TAC`, and `SIMP_TAC bool_ss []`. The second group consists of `DECIDE_TAC`, `ARITH_TAC`, and `REDUCE_TAC`. This also divides the procedures into those that handle pure logic and those that also handle arithmetic.

The tactics in the second group fail on almost of the proof obligations with essentially the same time efficiency. Below, we examine this phenomenon more closely.

The tactics in the first group prove all proof obligations valid with essentially the same time efficiency except for one on which they all fail. The results do not allow any clear conclusions, but they could indicate that `MESON_TAC` is less good at failing fast on goals it cannot prove. Due to the fact that the solving of proof obligation is inherently undecidable, we cannot expect to automatically decide all proof obligations. Instead, we hope that the decision procedures will solve a large percentage of the proof obligations. Therefore, it is important that a strategy quickly realizes whenever it cannot prove or disprove a goal.

Example	PO	TAUT			MESON			DECIDE			SIMP			ARITH			REDUCE		
		1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
Alarm	1	0.0	0.0	0.0	0.0	0.0	0.0	**0.0**	**0.0**	**0.0**	0.0	0.0	0.0	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
	2	0.0	0.0	0.0	0.0	0.0	0.0	**0.0**	**0.0**	**0.0**	0.0	0.0	0.0	0.0	0.0	0.0	**0.0**	**0.0**	**0.0**
	3	**0.0**	0.0	0.0	**0.0**	0.2	0.2	**0.0**	**0.1**	**0.1**	0.1	0.1	0.1	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
Gateway	1	0.0	0.0	0.0	0.0	0.0	0.0	**0.0**	**0.0**	**0.0**	0.0	0.0	0.0	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
	2	0.0	0.0	0.0	0.0	0.0	0.0	**0.0**	**0.0**	**0.0**	0.0	0.0	0.0	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
	3	0.0	0.0	0.0	0.0	0.0	0.0	**0.0**	**0.0**	**0.0**	0.0	0.0	0.0	0.0	0.0	0.0	**0.0**	**0.0**	**0.0**
Tracker	1	**0.1**	**0.1**	**0.1**	**20**	**21**	**22**	**0.1**	**0.2**	**2.5**	**0.0**	**0.1**	**0.3**	**0.1**	**0.2**	**0.6**	**0.0**	**0.1**	**0.2**
	3	0.0	0.1	0.1	0.1	0.2	0.4	**0.0**	**0.3**	**4.1**	0.0	0.1	0.4	**0.0**	**0.2**	**0.6**	**0.0**	**0.1**	**0.2**
	4	0.0	0.1	0.1	0.0	0.2	0.4	**0.0**	**0.2**	**3.3**	0.0	0.1	0.3	**0.0**	**0.2**	**0.6**	**0.0**	**0.1**	**0.1**
	5	**0.0**	0.1	0.1	**0.1**	0.3	0.5	**0.0**	**0.3**	**4.2**	**0.0**	0.1	0.4	**0.0**	**0.2**	**0.6**	**0.0**	**0.1**	**0.2**
	8	0.0	0.0	0.1	0.0	0.1	0.2	**0.0**	**0.2**	**3.9**	0.0	0.1	0.4	0.0	0.1	0.2	**0.0**	**0.1**	**0.2**
	9	**0.0**	**0.1**	0.2	**0.0**	**0.2**	0.5	**0.0**	**0.2**	**2.9**	**0.0**	**0.2**	0.3	**0.0**	**0.1**	**0.5**	**0.0**	**0.0**	**0.2**
	11	**0.0**	0.2	0.3	**0.1**	0.5	0.8	**0.0**	**0.4**	**11**	**0.0**	0.2	0.4	**0.0**	**0.3**	**0.7**	**0.0**	**0.1**	**0.2**
	13	**0.0**	0.1	0.1	**0.0**	0.1	0.2	**0.0**	**0.2**	**3.0**	**0.0**	0.1	0.3	**0.0**	0.1	0.2	**0.0**	**0.1**	**0.2**
	15	0.0	0.0	0.0	0.0	0.1	0.1	**0.0**	**0.1**	**0.1**	0.0	0.1	0.1	**0.0**	**0.1**	**0.1**	**0.0**	**0.0**	**0.0**
	16	0.0	0.0	0.0	0.0	0.1	0.1	**0.0**	**0.1**	**0.1**	0.0	0.1	0.1	**0.0**	**0.1**	**0.0**	**0.0**	**0.0**	**0.0**
	17	0.0	0.0	0.0	0.1	0.1	0.1	**0.0**	**0.1**	**0.1**	0.0	0.1	0.1	**0.0**	**0.1**	**0.1**	**0.0**	**0.0**	**0.0**

Table 3.1: Summary of experiments with domain checking proof obligations.

3.2.1 Unfolding definitions

In this section, we give examples of proof obligations proved valid using different degrees of unfolding of definitions.

No unfolding of specification definitions The proof obligation

```
val alarm_P01 = Term `!a per schedule.
 inv_Schedule(schedule) ==>
   (a IN alarms /\ per IN FDOM schedule ==>
     per IN FDOM schedule)`;
```

is an example of a formula solved without unfolding of the specification definitions. The proof obligation is generated to verify that the map lookup `schedule(per)` is done with an element `per` in the domain of the map `schedule`, that is, `per IN FDOM schedule`. The example illustrates a typically easy case where the map lookup is guarded by a condition `per IN FDOM schedule` ensuring exactly that the element is in the domain of the map.

Unfolding of projections and predicates The proof obligation

```
val tracker_P011 = Term `!trk cid ptoid pfromid.
        (inv_Tracker trk)  ==>
        (let pha = (mk_Phase(((
              Phase_contents(FAPPLY (Tracker_phases trk) ptoid)) UNION {cid}),
              Phase_expected_materials(FAPPLY (Tracker_phases trk) ptoid),
              Phase_capacity(FAPPLY (Tracker_phases trk) ptoid)))
         in
           pre_Move(trk,cid,ptoid,pfromid) ==>
           pre_Remove(trk,cid,pfromid))`;
```

is an example of a formula which is not solved without unfolding the specification definitions. Unfolding definitions of predicates (e.g. `pre_move`) and projections (e.g. `Phase_contents`), and then eliminating let-expressions yields the formula

```
``!trk cid ptoid pfromid.
    (!pid.
      pid IN FDOM (Tracker_phases trk) ==>
      FINITE (Phase_contents (FAPPLY (Tracker_phases trk) pid)) /\
      FINITE
        (Phase_expected_materials (FAPPLY (Tracker_phases trk) pid)) /\
      CARD (Phase_contents (FAPPLY (Tracker_phases trk) pid)) <=
      Phase_capacity (FAPPLY (Tracker_phases trk) pid) /\
      (Phase_expected_materials (FAPPLY (Tracker_phases trk) pid) <>
       {})) /\
    Consistent (Tracker_containers trk,Tracker_phases trk) /\
    PhasesDistinguished (Tracker_phases trk) /\
    MaterialSafe (Tracker_containers trk,Tracker_phases trk) ==>
    FINITE
      (Phase_contents
        (mk_Phase
          (Phase_contents (FAPPLY (Tracker_phases trk) ptoid) UNION {cid},
           Phase_expected_materials (FAPPLY (Tracker_phases trk) ptoid),
           Phase_capacity (FAPPLY (Tracker_phases trk) ptoid)))) /\
```

```
FINITE
  (Phase_expected_materials
    (mk_Phase
      (Phase_contents (FAPPLY (Tracker_phases trk) ptoid) UNION {cid},
       Phase_expected_materials (FAPPLY (Tracker_phases trk) ptoid),
       Phase_capacity (FAPPLY (Tracker_phases trk) ptoid)))) /\
CARD
  (Phase_contents
    (mk_Phase
      (Phase_contents (FAPPLY (Tracker_phases trk) ptoid) UNION {cid},
       Phase_expected_materials (FAPPLY (Tracker_phases trk) ptoid),
       Phase_capacity (FAPPLY (Tracker_phases trk) ptoid)))) <=
Phase_capacity
  (mk_Phase
    (Phase_contents (FAPPLY (Tracker_phases trk) ptoid) UNION {cid},
     Phase_expected_materials (FAPPLY (Tracker_phases trk) ptoid),
     Phase_capacity (FAPPLY (Tracker_phases trk) ptoid))) /\
(Phase_expected_materials
   (mk_Phase
     (Phase_contents (FAPPLY (Tracker_phases trk) ptoid) UNION {cid},
      Phase_expected_materials (FAPPLY (Tracker_phases trk) ptoid),
      Phase_capacity (FAPPLY (Tracker_phases trk) ptoid))) <>
 {}) ==>
Permission (trk,cid,ptoid) /\
pfromid IN FDOM (Tracker_phases trk) /\
cid IN Phase_contents (FAPPLY (Tracker_phases trk) pfromid) ==>
pfromid IN FDOM (Tracker_phases trk) /\
cid IN Phase_contents (FAPPLY (Tracker_phases trk) pfromid)``
```

which is solved. In fact, it would be enough to just unfold the definition of pre_Move. Also, it is worth mentioning that proof is in the last four lines of the listing above. The rest is not required.

Unfolding of all specification definitions The proof obligation

```
val tracker_PO9 = Term `!trk cid ptoid pfromid.
     (inv_Tracker trk) ==>
     (pre_Move(trk,cid,ptoid,pfromid) ==> ptoid IN (FDOM (Tracker_phases trk)))`;
```

is an example of a formula which is not solved when only projections and predicates are unfolded. Unfolding all definitions, and then eliminating let-expressions yields the formula

```
``!trk cid ptoid pfromid.
    (!pid.
      pid IN FDOM (Tracker_phases trk) ==>
      FINITE (Phase_contents (FAPPLY (Tracker_phases trk) pid)) /\
      FINITE
        (Phase_expected_materials (FAPPLY (Tracker_phases trk) pid)) /\
      CARD (Phase_contents (FAPPLY (Tracker_phases trk) pid)) <=
      Phase_capacity (FAPPLY (Tracker_phases trk) pid) /\
      (Phase_expected_materials (FAPPLY (Tracker_phases trk) pid) <>
       {})) /\
    (!pid.
      T ==>
```

```
   pid IN FDOM (Tracker_phases trk) ==>
   Phase_contents (FAPPLY (Tracker_phases trk) pid) SUBSET
   FDOM (Tracker_containers trk)) /\
 ~(?p1 p2.
    (T /\ T) /\
    p1 IN FDOM (Tracker_phases trk) /\
    p2 IN FDOM (Tracker_phases trk) /\
    (p1 <> p2) /\
    (Phase_contents (FAPPLY (Tracker_phases trk) p1) INTER
     Phase_contents (FAPPLY (Tracker_phases trk) p2) <>
     {})) /\
 (!pid.
   T ==>
   pid IN FDOM (Tracker_phases trk) ==>
   (!cid.
     T ==>
     cid IN Phase_contents (FAPPLY (Tracker_phases trk) pid) ==>
     cid IN FDOM (Tracker_containers trk) /\
     (cid IN FDOM (Tracker_containers trk) ==>
      Container_material (FAPPLY (Tracker_containers trk) cid) IN
      Phase_expected_materials (FAPPLY (Tracker_phases trk) pid)))) ==>
 (cid IN FDOM (Tracker_containers trk) /\
  ptoid IN FDOM (Tracker_phases trk) /\
  CARD (Phase_contents (FAPPLY (Tracker_phases trk) ptoid)) <
  Phase_capacity (FAPPLY (Tracker_phases trk) ptoid) /\
  Container_material (FAPPLY (Tracker_containers trk) cid) IN
  Phase_expected_materials (FAPPLY (Tracker_phases trk) ptoid)) /\
 pfromid IN FDOM (Tracker_phases trk) /\
 cid IN Phase_contents (FAPPLY (Tracker_phases trk) pfromid) ==>
 ptoid IN FDOM (Tracker_phases trk)''
```

which is solved. As in the case above, it would be enough to just unfold the definition of `pre_Move`. Also again, the proof is in the last nine lines of the listing above. The rest is not required.

For all three examples above, it is typical that the proof consists of "digging out" the consequence in the antecedent. Hence, the reasoning is essentially trivial.

3.2.2 Arithmetic

The decision procedures for arithmetic are not relevant for the proof obligations in our examples since no arithmetic occurs. Arithmetic is, however, important for domain checking proof obligations generated by arithmetic partial operators like division, and therefore it is also interesting to see to what extent and effeciency they cope with formulas without arithmetic. In this light it is disappointing that the decision procedures for arithmetic fails in almost all cases as reported in Table 3.1. However, the failures for a large part stem from the formulas not being in an appropriate normal form. For instance, ARITH_TAC solves the formula

```
val alarm_P02 =
  Term '!per plant. inv_Plant(plant) ==>
        (per IN FDOM (Plant_schedule plant) ==>
         per IN FDOM (Plant_schedule plant)) ';
```

but fails to solve the formula

```
val alarm_P01 = Term `!a per schedule.
 inv_Schedule(schedule) ==>
   (a IN alarms /\ per IN FDOM schedule ==>
     per IN FDOM schedule)`;
```

However, if the formula is put on implicative form

```
val alarm_P01_2 = Term `!a per schedule.
 inv_Schedule(schedule) ==>
   (a IN alarms ==> per IN FDOM schedule ==>
     per IN FDOM schedule)`;
```

ARITH_TAC solves it. A similar situation holds for REDUCE_TAC. For instance, the formula alarm_P01_2 defined above is reduced to

```
  ``!a per schedule. T``
```

by REDUCE_TAC but not solved. In fact, extending the tactic by first removing all universal quantifications from the front of a goal and then invoking REDUCE_TAC, that is, the tactic

```
   REPEAT GEN_TAC THEN REDUCE_TAC
```

yields a tactic which solves as many or rather as few of the proof obligations in Table 3.1 as ARITH_TAC with essentially the same efficiency.

An obvious approach would hence be to first move formulas into an appropriate normal form and then apply the respective decision procedures. However, the appropriateness of the approach seems hard to justify by the collection of proof obligations available in this case study. Because even mild rewriting is on its own sufficient to solve almost all domain checking proof obligations in the examples. In particular, removing the PURE prefix from the tactics above

```
fun VDM_PROJ_AND_PRED_DEF_RW (tac:tactic):tactic =
  RW_TAC (append proj_DEF_rules pred_DEF_rules) THEN
  (CONV_TAC (DEPTH_CONV let_CONV)) THEN tac;

fun VDM_DEF_RW (tac:tactic):tactic =
  RW_TAC DEF_rules THEN
  (CONV_TAC (DEPTH_CONV let_CONV)) THEN tac;
```

that is, allowing RW_TAC to apply its build–in rewrites yields tactics which solve most of the domain checking proof obligations before invoking the argument tactic tac. Moreover, also the standard tactic

```
REPEAT STRIP_TAC
```

solves almost all the domain checking proof obligations as efficiently as TAUT_TAC. This fact supports the observation from above that the proofs often consists of digging out the consequence in the antecedent.

For the decision procedure implemented by DECIDE_TAC, the problem seems to be that the tactic, while not properly programmed with definitions for sets and maps, does not interpret constants like IN and FDOM as uninterpreted constants. In fact, DECIDE_TAC proves many[2] of the proof obligations when read in uninterpreted, that is, without reading in definitions of sets, lists, maps, and user defined things. However, decisionLib admits facilities for programming DECIDE_TAC to work with new types and also to extend it with other decision procedures. What we have tried out is only the raw version of DECIDE_TAC. Further investigations are needed.

3.2.3 Simplification

One domain checking proof obligation, tracker_P01, is not proved valid by any of the decision strategies. The property verified by tracker_P01 is that the mappings applied to the merge map operator, *munion*, are compatible, see [2], that is, that each element belonging to the domain of both mappings is mapped to the same element by both mappings

```
val tracker_P01 = Term `! trk cid quan mat.
     inv_Tracker trk ==>
     pre_Introduce(trk,cid,quan,mat)  ==>
     !c1 c2.
         c1 IN (FDOM (Tracker_containers trk)) /\
         c2 IN (FDOM (MAPENUM [(cid,mk_Container(quan, mat))]))
       ==>
         (c1 = c2) ==>
            ((FAPPLY (Tracker_containers trk) c1) =
             (FAPPLY (MAPENUM [(cid,mk_Container(quan, mat))]) c2))`;
```

After inspecting tracker_P01, it is no surprise that it cannot be solved by a procedure for propositional logic. The expression

```
 MAPENUM [(cid,mk_Container(quan, mat))]
```

defines a map enumeration, here a mapping with the singleton domain consisting of cid. In particular in the case of tracker_P01, c2 is equal to cid and c1 which is the essential information in the proof of its validity. Since this information is extractable by rewriting, it would be tempting to try to solve it using the tactic

```
VDM_DEF_RW (SIMP_TAC vdm_ss vdm_rewrites)
```

The simplification set (ss) and rewrite theorems used here are described in the code [4]. However, the result is the following subgoal

```
    ``!trk cid quan mat c1 c2.
        (!pid.
          pid IN FDOM (Tracker_phases trk) ==>
          FINITE (Phase_contents (FAPPLY (Tracker_phases trk) pid))) /\
        (!pid.
```

[2]Many, because not all proof obligations can be read in uninterpreted, for instance the set enumeration notation {...} is not accepted by the default syntax checker

```
  pid IN FDOM (Tracker_phases trk) ==>
  FINITE
    (Phase_expected_materials (FAPPLY (Tracker_phases trk) pid))) /\
(!pid.
  pid IN FDOM (Tracker_phases trk) ==>
  CARD (Phase_contents (FAPPLY (Tracker_phases trk) pid)) <=
  Phase_capacity (FAPPLY (Tracker_phases trk) pid)) /\
(!pid.
  pid IN FDOM (Tracker_phases trk) ==>
  ({} <>
   Phase_expected_materials (FAPPLY (Tracker_phases trk) pid))) /\
(!pid.
  pid IN FDOM (Tracker_phases trk) ==>
  Phase_contents (FAPPLY (Tracker_phases trk) pid) SUBSET
  FDOM (Tracker_containers trk)) /\
(!p1 p2.
  p1 IN FDOM (Tracker_phases trk) /\
  p2 IN FDOM (Tracker_phases trk) /\
  (p1 <> p2) ==>
  (Phase_contents (FAPPLY (Tracker_phases trk) p1) INTER
   Phase_contents (FAPPLY (Tracker_phases trk) p2) =
   {})) /\
(!pid cid.
  pid IN FDOM (Tracker_phases trk) /\
  cid IN Phase_contents (FAPPLY (Tracker_phases trk) pid) ==>
  cid IN FDOM (Tracker_containers trk)) /\
(!pid cid.
  pid IN FDOM (Tracker_phases trk) /\
  cid IN Phase_contents (FAPPLY (Tracker_phases trk) pid) /\
  cid IN FDOM (Tracker_containers trk) ==>
  Container_material (FAPPLY (Tracker_containers trk) cid) IN
  Phase_expected_materials (FAPPLY (Tracker_phases trk) pid)) /\
~(cid IN FDOM (Tracker_containers trk)) /\
c1 IN FDOM (Tracker_containers trk) /\
(c2 = cid) /\
(c1 = c2) ==>
(FAPPLY (Tracker_containers trk) cid = mk_Container (quan,mat))``
```

showing that the simplifier fails to identify that `c1` and `cid` are equal, and hence fails to catch the contradiction in the antecedents. The problem is the ordering of the assumptions. Below, we rephrase the problem and give a partial solution. First, we split the goal using `VDM_SPLIT_TAC` to get

```
``FAPPLY (Tracker_containers trk) cid = mk_Container (quan,mat)``
------------------------------------
    ``c1 = c2``
    ``c2 = cid``
    ``c2 IN FDOM (Tracker_containers trk)``
    ``~(cid IN FDOM (Tracker_containers trk))``

    ....

    ``!pid.
        pid IN FDOM (Tracker_phases trk) ==>
        FINITE (Phase_contents (FAPPLY (Tracker_phases trk) pid))``
```

An invocation of the full simplification tactic with simpset bool_ss and no rewrite theorems, that is, the tactic

```
FULL_SIMP_TAC bool_ss []
```

leaves the goal invariant. This is because the FULL_SIMP_TAC tactic simplifies each assumption (bottom-up in the ordering of the listing above) using the assumptions already simplified as extra rewrite theorems. Hence, the ordering of assumptions can be essential as in this case where the assumption

```
``~(cid IN FDOM (Tracker_containers trk))``
```

is simplified without using the assumptions

```
``c1 = cid``
``c2 = cid``
``c2 IN FDOM (Tracker_containers trk)``
```

We have modified the standard full simplification tactic to help avoid some of the problems with the dependency on the ordering of assumptions. Our version, called OUR_FULL_SIMP_TAC, works as FULL_SIMP_TAC except that when simplifying an assumption it uses *all other assumptions*; those that have already been simplified as well as those to be simplified. Hence, the modification is not ideal since it depends on the ordering of assumptions but less than the standard version. In particular, invoking OUR_FULL_SIMP_TAC bool_ss [] solves the goal from above. Altogether, the tactic

```
VDM_DEF_RW (SIMP_TAC vdm_ss vdm_rewrites) THEN
VDM_SPLIT_TAC THEN (OUR_FULL_SIMP_TAC bool_ss [])
```

proves the tracker_PO1 in 2.7 seconds.

Chapter 4

Subtype checking

The second class of proof obligations that we investigate is the class generated from the use of subtyping.

In the examples [5, 3], the proof obligations 2,6,7,10,12, and 14 of the tracker example are classified as subtype checking proof obligations.

4.1 Strategies

Our main interest is the investigation of classes of proof obligations that are in practice equally hard or easy to solve, i.e., that are in most cases solvable by the same techniques.

A first naive attempt could be to try out the strategies that worked for domain checking proof obligations in Chapter 3. But none of the subtype checking proof obligations are proved by the decision procedures discussed in Chapter 3.

Instead, we suggest a rather brute-force strategy involving both

- simplification,
- goal directed reasoning, and
- resolution.

The overall strategy that we apply is goal directed reasoning combined with simplification, and when everything else fails we then apply resolution.

The main tactic

```
fun VDM_REC_INV_TAC (depth:int):tactic = fn g => (
  let val _ = max_depth := (if depth > (!max_depth) then depth
                                                    else (!max_depth))
  in
    VDM_SPLIT_TAC THEN
    ((SIMP_TAC vdm_arith_ss vdm_rewrites) THEN
    FIRST
      [OUR_ASM_ACCEPT_TAC,
       COND_CASES_TAC THEN (VDM_REC_INV_TAC (depth+1)),
```

```
      (VDM_INV_EXISTS_TAC (VDM_REC_INV_TAC (depth+1))),
      VDM_CONTRADICT_TAC,
      (VDM_DEDUCT_TAC (VDM_REC_INV_TAC (depth+1))),
      (VDM_SIMP_TAC (VDM_REC_INV_TAC (depth+1))),
      (VDM_RES_TAC NO_TAC),
      (BACKTRACKING_TAC depth)
     ]
   )
  end
) g;
```

is a depth-first recursive tactic performing depth-first proof search. The tactic was constructed gradually in an "extend by need" fashion. While solving the proof obligations one by one, we extended the current tactic with new techniques and theorems when subgoals were not proved by the current tactic. Much effort has gone into adding "good" rewrite theorems to the simplifier both theorems from the standard libraries and new theorems combining reasoning about sets, maps, and lists. This means that the tactic is somewhat tailored towards proving the proof obligations of the examples and therefore still rather unpolished, unoptimized, and primitive.

Below, we give a short description of each of the component tactics.

VDM_SPLIT_TAC first splits the goal using `REPEAT STRIP_TAC` then applies a case analysis tactic on conditionals in the assumptions, and then performs simple term substitutions in the assumptions based on equalities in the assumptions.

SIMP_TAC vdm_arith_ss vdm_rewrites simplifies the conclusion of the goal using the largest simpset that we use, `vdm_arith_ss`, and the largest list of rewrite theorems that we use, `vdm_rewrites`.

OUR_ASM_ACCEPT_TAC tries to solve the goal by applying simple checks for acceptance by matching each of the assumptions against the conclusion.

COND_CASES_TAC does a case analysis on the conclusion of the goal.

VDM_INV_EXISTS_TAC (VDM_REC_INV_TAC (depth+1)) fails if the conclusion of the goal is not existentially quantified, and otherwise invokes the tactic `OUR_INV_EXISTS_TAC` followed by a recursive application of `VDM_REC_INV_TAC`. On failure it first simplifies the goal, then performs resolution, using the tactic `VDM_RES_TAC` described below, and then invokes tactic `OUR_INV_EXISTS_TAC` followed by a recursive application of `VDM_REC_INV_TAC`. The tactic `OUR_INV_EXISTS_TAC`, is a simple tactic for solving existentially quantified goals. Given a goal of the form

```
A ?- ?x1...?xm. C1 /\ ... /\ Cn
```

it first looks for instantiations of the variables by matching for each of the clauses individually against each of the assumptions, and then invokes the argument tactic on the instantiated goal trying out all of the instantiations found taking the one that lead to the most matches first.

VDM_CONTRADICT_TAC fails if conclusion of the goal is not Boolean constant F, and otherwise tries to prove a contradiction in the assumptions by simplification followed by resolution. There is no recursive call. The resolution is performed by the tactic VDM_RES_TAC described below.

VDM_DEDUCT_TAC (VDM_REC_INV_TAC (depth+1)) does goal directed reasoning MATCH_MP_TAC applied to theorems and assumptions. If a match results in an exists goal then the tactic OUR_INV_EXISTS_TAC described above is invoked.

VDM_SIMP_TAC (VDM_REC_INV_TAC (depth+1)) simplifies the goal. If the simplification does not make any changes then it fails otherwise it applies VDM_REC_INV_TAC recursively. The simplification divides into two; first simplification is applied only to the goal and to the assumptions that do not begin with a quantifier, if this simplification fails to make any changes, a second simplification is applied this time simplifying the goal and all of the assumptions.

VDM_RES_TAC applies resolution followed by simplification in three ways

```
fun VDM_RES_TAC (tac:tactic):tactic =
  let val _ = vdm_res_tac := (!vdm_res_tac) + 1 in
  FIRST
   [RES_TAC THEN (ASM_SIMP_TAC vdm_ss vdm_rewrites) THEN tac,
    VDM_PRED_SET_EXT_TAC THEN
    RES_TAC THEN (OUR_SIMP_ASM_TAC vdm_ss vdm_rewrites) THEN tac,
    (MAP_EVERY ASSUME_TAC res_list) THEN
    VDM_PRED_SET_EXT_TAC THEN
    RES_TAC THEN (OUR_SIMP_ASM_TAC vdm_ss vdm_rewrites) THEN tac
   ]
  end;
```

First it applies resolution on the assumptions, using the standard tactic RES_TAC, followed by simplification of the conclusion using the assumptions. If the first attempt fails it rewrites the assumptions with some special rewrite theorems before applying RES_TAC and simplification. Examples of the special rewrite theorems used are

```
|- !s t. s SUBSET t = (!x. x IN s ==> x IN t)
```

and

```
|- !s. (s = {}) = (!x. ~(x IN s))
```

They are intended to lower the reasoning about sets to the element level.

If also the second attempt fails it first adds a list of theorems to the assumptions and then does as in the second attempt.

Example	PO	Duration (sec.)	vdm_split_tac	vdm_inv_exists_tac	vdm_contradict_tac	vdm_deduct_tac	vdm_simp_tac	vdm_res_tac	max_depth	backtracking	max_subgoals
Tracker	2	6.6	5	6	6	5	5	5	2	5	4
	6	12.2	4	7	7	4	4	4	2	4	4
	7	101.3	23	48	48	23	23	39	3	23	34
	10	8.5	1	4	4	1	1	1	0	1	4
	12	528.5	44	139	139	44	44	125	4	44	114
	14	127.1	23	56	56	23	23	47	3	23	42

Table 4.1: Summary of experiments with subtype checking proof obligations.

BACKTRACKING_TAC simply prints "backtracking" on the screen and then fails if the depth if the depth is greater than zero and otherwise behaves as the identity tactic. The test for zero makes sure that the tactic backtracks appropriately but also returns subgoals not proved.

```
val VDM_RW_DEF_TAC:tactic =
  (CONV_TAC (PURE_RW_CONV DEF_rules  THENC
             (DEPTH_CONV let_CONV)  THENC
             (RW_CONV vdm_rewrites)));
```

Before invoking the recursive invariants tactic we rewrite all specification definitions using the tactic

```
val VDM_RW_DEF_TAC:tactic =
  (CONV_TAC (PURE_RW_CONV DEF_rules  THENC
             (DEPTH_CONV let_CONV)  THENC
             (RW_CONV vdm_rewrites)));
```

Hence, the invariants tactic we suggest is

```
val VDM_INV_TAC  = VDM_RW_DEF_TAC THEN (VDM_REC_INV_TAC 0);
```

4.2 Summary of experiments

In this section, we summarize the experiments that we performed with the solving of subtype checking proof obligations.

The file vdm-varians-tactics.sml containing the HOL98 tactics used, and the file tracker-proof-lab.sml containing the applications of the tactics are found in [4].

Table 4.1 summarizes the results obtained. We have applied the invariants tactic to each of the proof obligations. All proof obligations were solved. The Duration column marks the response time in seconds, The max_subgoals column marks the maximal number of immediate subgoals resulting from an application of a tactic component tactic in a THEN-sequencing of tactics, that is, the maximal number of immediate subgoals passed to the tactical THEN during the proof. The max_depth column marks the maximal recursion depth reached by VDM_REC_INV_TAC during the proof. The rest of the columns mark the number of invocations of the respective component tactics.

Simplification on its own can solve a proof obligation like

```
val tracker_PO10 = Term `!trk cid ptoid pfromid.
     (inv_Tracker trk) ==>
     (pre_Move(trk,cid,ptoid,pfromid) ==>
      inv_Phase(mk_Phase(((Phase_contents(FAPPLY (Tracker_phases trk) ptoid)) UNION {cid}),
       Phase_expected_materials(FAPPLY (Tracker_phases trk) ptoid),
       Phase_capacity(FAPPLY (Tracker_phases trk) ptoid))))`;
```

Resolution versus simplification Note that in the invariants tactic above we try to make sure that resolution is not followed by simpification of assumptions using the assumptions as rewrite theorems because such a simplification would cancel the consequences derived by the resolution. Consider the goal

```
``g``
------------------------------------
    ``p x``
    ``!y. p y ==> q y``
```

invoking RES_TAC yields the goal

```
``g``
------------------------------------
    ``p x``
    ``!y. p y ==> q y``
    ``q x``
```

with the derived assumption q x which is then canceled by an invocation of OUR_FULL_SIMP_TAC bool_ss [] which yields the goal

```
``g``
------------------------------------
    ``p x``
    ``!y. p y ==> q y``
    ``T``
```

Resolution should be used with care. Consider the proof obligation

```
val tracker_P014 = Term `!trk cid source.
    (inv_Tracker trk) ==>
    (pre_Delete (trk,cid,source) ==>
     inv_Tracker(mk_Tracker(({cid} <-: (Tracker_containers trk)),
               Tracker_phases(Remove(trk,cid,source)))))`;
```

Proving tracker_P014 valid includes solving subgoals like the one shown in Figure 4.1 Such a goal is difficult to solve using goal directed reasoning since a contradiction in the assumptions needs to be derived. An obvious next move would be to use resolution. This should however only be done with care as shown by the subgoal in Figure 4.2 resulting from an invocation of RES_TAC. Hopefully, we will never have to present any of these goals to an end-user!

Arithmetic is needed in order to prove tracker_P010 valid. If ARITH_ss is removed from the vdm_ss simpset then the VDM_INV_TAC tactic invoked on tracker_P010 cannot solve the subgoal

```
  ``1 + (CARD (Phase_contents (FAPPLY (Tracker_phases trk) ptoid)) -
    CARD ({cid} INTER Phase_contents (FAPPLY (Tracker_phases trk) ptoid))) <=
     Phase_capacity (FAPPLY (Tracker_phases trk) ptoid)``
   ____________________________________
       ``cid IN Phase_contents (FAPPLY (Tracker_phases trk) pfromid)``
       ``pfromid IN FDOM (Tracker_phases trk)``
       ``Container_material (FAPPLY (Tracker_containers trk) cid) IN
         Phase_expected_materials (FAPPLY (Tracker_phases trk) ptoid)``
       ``CARD (Phase_contents (FAPPLY (Tracker_phases trk) ptoid)) <
         Phase_capacity (FAPPLY (Tracker_phases trk) ptoid)``
       ``ptoid IN FDOM (Tracker_phases trk)``
       ``cid IN FDOM (Tracker_containers trk)``
       ``!pid.
           pid IN FDOM (Tracker_phases trk) ==>
           (!cid.
             cid IN Phase_contents (FAPPLY (Tracker_phases trk) pid) ==>
             cid IN FDOM (Tracker_containers trk) /\
             (cid IN FDOM (Tracker_containers trk) ==>
              Container_material (FAPPLY (Tracker_containers trk) cid) IN
              Phase_expected_materials (FAPPLY (Tracker_phases trk) pid)))``
       ``~(?p1 p2.
            p1 IN FDOM (Tracker_phases trk) /\
            p2 IN FDOM (Tracker_phases trk) /\
            (p1 <> p2) /\
            (Phase_contents (FAPPLY (Tracker_phases trk) p1) INTER
             Phase_contents (FAPPLY (Tracker_phases trk) p2) <>
             {}))``
       ``!pid.
           pid IN FDOM (Tracker_phases trk) ==>
           Phase_contents (FAPPLY (Tracker_phases trk) pid) SUBSET
           FDOM (Tracker_containers trk)``
       ``!pid.
           pid IN FDOM (Tracker_phases trk) ==>
           FINITE (Phase_contents (FAPPLY (Tracker_phases trk) pid)) /\
           FINITE
```

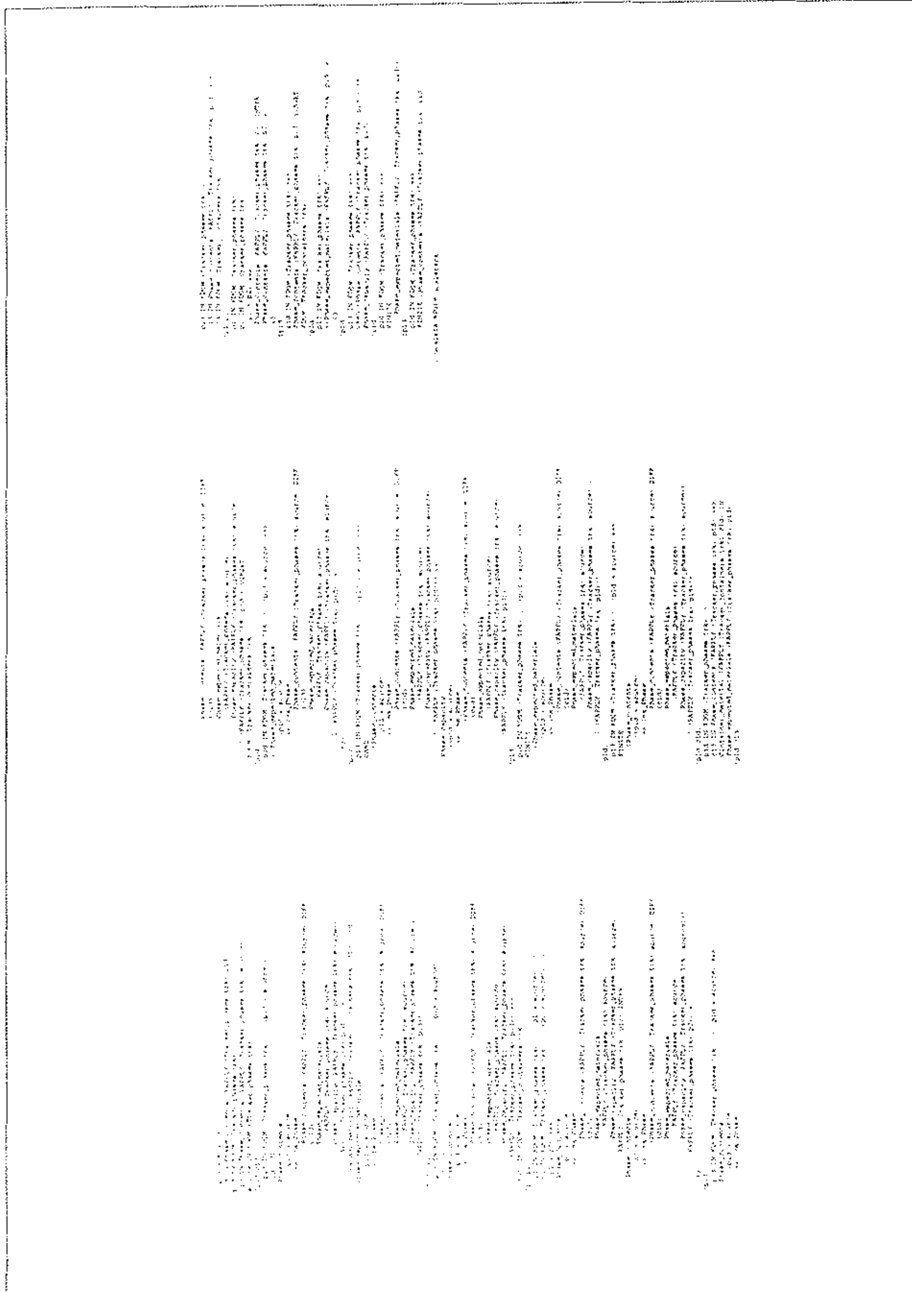

Figure 4.1: A subgoal arising in proving PO14 in the tracker example.

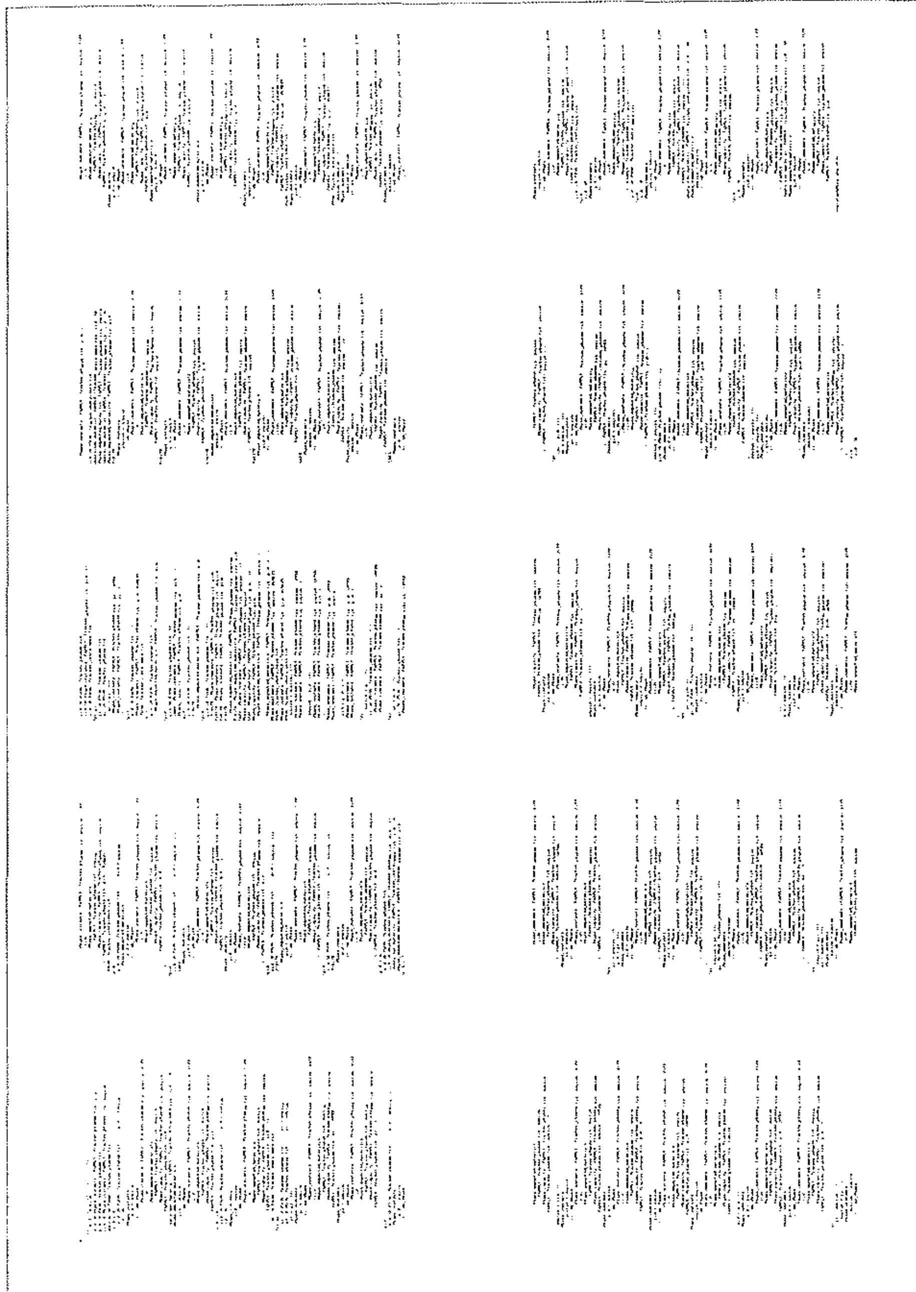

Figure 4.2: After invoking RES_TAC on the subgoal in Figure 4.1.

```
  (Phase_expected_materials (FAPPLY (Tracker_phases trk) pid)) /\
CARD (Phase_contents (FAPPLY (Tracker_phases trk) pid)) <=
Phase_capacity (FAPPLY (Tracker_phases trk) pid) /\
(Phase_expected_materials (FAPPLY (Tracker_phases trk) pid) <>
 {})``
```

In fact, all of the subtype checking proof obligations except `tracker_PO2` require arithmetic.

4.2.1 Reusing type information

Sometimes type correctness stated in a proof obligation can – if proved – be reused when proving other proof obligations. The reason for doing this is the hope of gaining effeciency by not having to redo proofs. In this section, we investigate three examples taken from the tracker example. Consider the proof obligation `tracker_PO7` from the tracker example. The proof obligation states that the type of the expression

```
mk_Tracker(containers,(phases MOVERRIDE (MAPENUM [(source, pha)])))
```

is a subtype of `Tracker`. For this to be the case, the variable `pha` needs to be of type `Phase`. But, this is exactly the property stated in `tracker_PO6`. Under the assumption that `tracker_PO6` is proved it would suffice to prove the following modified version of `tracker_PO7`

```
val tracker_PO7_2 = Term `!containers phases cid source.
        inv_PhaseInfo(phases) ==>
        inv_Tracker(mk_Tracker(containers,phases)) ==>
        (let pha = (mk_Phase(((Phase_contents(FAPPLY  phases source)) DIFF {cid}),
                   Phase_expected_materials(FAPPLY  phases source),
                   Phase_capacity(FAPPLY phases source)))
        in
        inv_Phase(pha) ==>
        pre_Remove(mk_Tracker(containers,phases),cid,source)
        ==>
        inv_Tracker(mk_Tracker(containers,
                (phases MOVERRIDE (MAPENUM [(source, pha)])))))`;
```

where we have added the assumption that `pha` satisfies the `Phase` invariant. Similarly, we can reuse the type correctness of the `Remove` function stated in `tracker_PO7` (`tracker_PO7_2`) to modify `tracker_PO14` to:

```
val tracker_PO14_2 = Term `!trk cid source.  (inv_Tracker trk) ==>
    inv_Tracker(Remove(trk,cid,source)) ==>
    (pre_Delete (trk,cid,source) ==>
     inv_Tracker(mk_Tracker(({cid} <-: (Tracker_containers trk)),
             Tracker_phases(Remove(trk,cid,source)))))`;
```

Also, `tracker_PO12` can be modified using the type correctness of `Remove` and `pha` stated in respectively `tracker_PO7` (`tracker_PO7_2`) and `tracker_PO14` (`tracker_PO14_2`):

Example	PO	Duration (sec.)	vdm_split_tac	vdm_inv_exists_tac	vdm_contradict_tac	vdm_deduct_tac	vdm_simp_tac	vdm_res_tac	max_depth	backtracking	max_subgoals
Tracker	7	101.3	23	48	48	23	23	39	3	23	34
	7_2	112.1	19	40	40	19	19	35	3	19	34
	12	528.5	44	139	139	44	44	125	4	44	114
	12_2	1681.4	44	133	133	44	44	125	4	44	114
	14	127.1	23	56	56	23	23	47	3	23	42
	14_2	122.7	1	16	16	1	1	9	0	0	16

Table 4.2: Summary of experiments on reuse of extra type information.

```
val tracker_PO12_2 = Term `!trk cid ptoid pfromid.
        (inv_Tracker trk) ==>
        (let pha = (mk_Phase(((
              Phase_contents(FAPPLY (Tracker_phases trk) ptoid)) UNION {cid}),
              Phase_expected_materials(FAPPLY (Tracker_phases trk) ptoid),
              Phase_capacity(FAPPLY (Tracker_phases trk) ptoid)))
         in
         inv_Tracker(Remove(trk,cid,pfromid)) ==>
         inv_Phase(pha) ==>
         pre_Move(trk,cid,ptoid,pfromid) ==>
           inv_Tracker(mk_Tracker(Tracker_containers trk,
               (Tracker_phases(Remove(trk,cid,pfromid)) MOVERRIDE
                        MAPENUM [(ptoid,pha)]))))`;
```

Table 4.2 summarizes the results obtained by applying the invariants tactic to each of the modified proof obligation. The table should be read as explained for Table 4.1 in Section 4.2. All proof obligations were proved valid. There seems to be no clear conclusions; for `tracker_PO12` there is a dramatic loss in time efficiency and for `tracker_PO14` is possibly a slight gain in time efficiency. One interesting observation however seems to be that the modified versions are solved using fewer tactic invocations. In particular, `tracker_PO14_2` seems to have a nice flat proof structure with no recursive calls, and hence no backtracking. Though no conclusion can be made, the result might indicate that the size of the goals is more important to the time efficiency than the number of invocations on tactics.

4.2.2 Normal form

In this section, we adress the question of why did we not simplify, and hence normalize, our goal before calling the recursive tactic `VDM_REC_INV_TAC`? The

Example	PO	Duration (sec.)	vdm_split_tac	vdm_inv_exists_tac	vdm_contradict_tac	vdm_deduct_tac	vdm_simp_tac	vdm_res_tac	max_depth	backtracking	max_subgoals
Tracker	2	10.8	6	5	5	6	6	6	2	6	3
	6	11.3	3	3	3	3	3	3	1	3	2
	7	98.7	49	43	43	49	49	49	3	49	22
	10	6.9	1	1	1	1	1	1	0	1	1
	12	537.9	146	121	121	146	146	152	5	146	42
	14	**88.7**	3	9	9	3	3	3	1	3	30

Table 4.3: Summary of experiments with normalising first.

answer illustrates the subtleties of automated theorem proving. A seemingly obvious suggestion turns out to have quite divergent consequences on distinct goals. Some are proved faster, others slower, and yet others are no longer proved. Table 4.2 summarizes the results obtained by applying the invariant tactic to each of the modified proof obligation. The table should be read as explained for Table 4.1 in Section 4.2. The grey shaded box in Table 4.3 marks that the strategy failed. The proof obligation `tracker_PO14` is not proved by the new strategy. Instead, it results in one remaining subgoal:

```
``cid <> cid'``
____________________________________
    ``~(pid = source)``
    ``cid' IN Phase_contents (FAPPLY (Tracker_phases trk) pid)``
    ``pid IN FDOM (Tracker_phases trk)``
    ``cid IN Phase_contents (FAPPLY (Tracker_phases trk) source)``
    ``source IN FDOM (Tracker_phases trk)``
    ``!pid cid.
        pid IN FDOM (Tracker_phases trk) /\
        cid IN Phase_contents (FAPPLY (Tracker_phases trk) pid) /\
        cid IN FDOM (Tracker_containers trk) ==>
        Container_material (FAPPLY (Tracker_containers trk) cid) IN
        Phase_expected_materials (FAPPLY (Tracker_phases trk) pid)``
    ``!pid cid.
        pid IN FDOM (Tracker_phases trk) /\
        cid IN Phase_contents (FAPPLY (Tracker_phases trk) pid) ==>
        cid IN FDOM (Tracker_containers trk)``
    ``!p1 p2.
        p1 IN FDOM (Tracker_phases trk) /\
        p2 IN FDOM (Tracker_phases trk) /\
        (p1 <> p2) ==>
        (Phase_contents (FAPPLY (Tracker_phases trk) p1) INTER
```

```
     Phase_contents (FAPPLY (Tracker_phases trk) p2) =
     {})``
``!pid.
    pid IN FDOM (Tracker_phases trk) ==>
    Phase_contents (FAPPLY (Tracker_phases trk) pid) SUBSET
    FDOM (Tracker_containers trk)``
``!pid.
    pid IN FDOM (Tracker_phases trk) ==>
    ({} <>
     Phase_expected_materials (FAPPLY (Tracker_phases trk) pid))``
``!pid.
    pid IN FDOM (Tracker_phases trk) ==>
    CARD (Phase_contents (FAPPLY (Tracker_phases trk) pid)) <=
    Phase_capacity (FAPPLY (Tracker_phases trk) pid)``
``!pid.
    pid IN FDOM (Tracker_phases trk) ==>
    FINITE
      (Phase_expected_materials (FAPPLY (Tracker_phases trk) pid))``
``!pid.
    pid IN FDOM (Tracker_phases trk) ==>
    FINITE (Phase_contents (FAPPLY (Tracker_phases trk) pid))``
```

The example illustrates just how fragile a general strategy like the above is to changes in the exact representation of the goal. We shall not go into exactly why the strategy fails to prove the goal. Instead, we illustrate the sort of the reasoning required. We need to show that

```
``cid <> cid'``
```

First, we notice that `cid` and `cid'` belong to sets

```
``cid' IN Phase_contents (FAPPLY (Tracker_phases trk) pid)``
``cid IN Phase_contents (FAPPLY (Tracker_phases trk) source)``
```

Hence, we are done if we can show that the sets are disjoint. Second, the sets are in fact disjoint since

```
``~(pid = source)``
``pid IN FDOM (Tracker_phases trk)``
``source IN FDOM (Tracker_phases trk)``
```

and

```
``!p1 p2.
    p1 IN FDOM (Tracker_phases trk) /\
    p2 IN FDOM (Tracker_phases trk) /\
    (p1 <> p2) ==>
    (Phase_contents (FAPPLY (Tracker_phases trk) p1) INTER
     Phase_contents (FAPPLY (Tracker_phases trk) p2) =
     {})``
```

4.2.3 Translational choices

In the translation of VDM-SL specifications into HOL98 specifications, we have experimented with

- the translation of A and B into `A /\ (A ==> B)`, and
- with the elimination of quantifications over ranges.

Below, we test how well the invariants tactic does if the VDM-SL and is translated directly into the HOL98 /\, and if the quantification over ranges is kept.

Andalso Consider the function

```
val MaterialSafe_DEF = func `MaterialSafe(containers,phases) =
  !pid. pid IN FDOM phases ==>
    !cid. cid IN (Phase_contents (FAPPLY phases pid)) ==>
      (cid IN FDOM containers /\
      (cid IN FDOM containers  ==>
      (Container_material(FAPPLY containers cid) IN
      (Phase_expected_materials (FAPPLY phases pid)))))`;
```

from the tracker example. Now, consider the modified tracker example with the alternative definition of `MaterialSafe`

```
val MaterialSafe2_DEF = func `MaterialSafe(containers,phases) =
  !pid. pid IN FDOM phases ==>
    !cid. cid IN (Phase_contents (FAPPLY phases pid)) ==>
      (cid IN FDOM containers /\
      (Container_material(FAPPLY containers cid) IN
      (Phase_expected_materials (FAPPLY phases pid))))`;
```

where

```
  (cid IN FDOM containers  ==>
  (Container_material(FAPPLY containers cid) IN
  (Phase_expected_materials (FAPPLY phases pid))))
```

has been replaced by

```
  (Container_material(FAPPLY containers cid) IN
  (Phase_expected_materials (FAPPLY phases pid)))
```

Table 4.4 summarizes the results obtained by applying the invariants tactic to each of the proof obligations. The table should be read as explained for Table 4.1 in Section 4.2. The grey shaded box marks that the strategy failed.

All proof obligations but `tracker_PO14` are proved valid with slightly better time efficiency but otherwise with the almost the same numbers as in Table 4.1. Instead, `tracker_PO14` results in one remaining subgoal

```
    ``Container_material
        (FAPPLY
           (DRESTRICT (Tracker_containers trk)
              (FDOM (Tracker_containers trk) DIFF {cid}))
          cid') IN
      Phase_expected_materials (FAPPLY (Tracker_phases trk) pid)``
```

Example	PO	Duration (sec.)	vdm_split_tac	vdm_inv_exists_tac	vdm_contradict_tac	vdm_deduct_tac	vdm_simp_tac	vdm_res_tac	max_depth	backtracking	max_subgoals
Tracker	2	4.8	5	5	5	5	5	5	2	5	3
	6	11.9	4	7	7	4	4	4	2	4	4
	7	97.8	23	48	48	23	23	39	3	23	34
	10	8.1	1	4	4	1	1	1	0	1	4
	12	503.8	44	139	139	44	44	125	4	44	114
	14	**159.6**	26	59	59	26	26	50	3	26	42

Table 4.4: Summary of experiments on tracker without andalso.

```
-----------------------------------
    ``~(pid = source)``
    ``cid' IN Phase_contents (FAPPLY (Tracker_phases trk) pid)``
    ``pid IN FDOM (Tracker_phases trk)``
    ``cid IN Phase_contents (FAPPLY (Tracker_phases trk) source)``
    ``source IN FDOM (Tracker_phases trk)``
    ``!pid.
        pid IN FDOM (Tracker_phases trk) ==>
        (!cid.
          cid IN Phase_contents (FAPPLY (Tracker_phases trk) pid) ==>
          cid IN FDOM (Tracker_containers trk) /\
          Container_material (FAPPLY (Tracker_containers trk) cid) IN
          Phase_expected_materials (FAPPLY (Tracker_phases trk) pid))``
    ``~(?p1 p2.
         p1 IN FDOM (Tracker_phases trk) /\
         p2 IN FDOM (Tracker_phases trk) /\
         (p1 <> p2) /\
         (Phase_contents (FAPPLY (Tracker_phases trk) p1) INTER
          Phase_contents (FAPPLY (Tracker_phases trk) p2) <>
          {}))``
    ``!pid.
        pid IN FDOM (Tracker_phases trk) ==>
        Phase_contents (FAPPLY (Tracker_phases trk) pid) SUBSET
        FDOM (Tracker_containers trk)``
    ``!pid.
        pid IN FDOM (Tracker_phases trk) ==>
        FINITE (Phase_contents (FAPPLY (Tracker_phases trk) pid)) /\
        FINITE
          (Phase_expected_materials (FAPPLY (Tracker_phases trk) pid)) /\
        CARD (Phase_contents (FAPPLY (Tracker_phases trk) pid)) <=
        Phase_capacity (FAPPLY (Tracker_phases trk) pid) /\
        (Phase_expected_materials (FAPPLY (Tracker_phases trk) pid) <>
         {})``
```

To solve the subgoal, we could first deduce the `cid <> cid'` like in the proof at the end of the previous section. Once, we know this the goal simplifies to

```
''Container_material
    (FAPPLY (Tracker_containers trk) cid') IN
  Phase_expected_materials (FAPPLY (Tracker_phases trk) pid)''
```

Using the assumptions

```
''cid' IN Phase_contents (FAPPLY (Tracker_phases trk) pid)''
''pid IN FDOM (Tracker_phases trk)''
''!pid.
    pid IN FDOM (Tracker_phases trk) ==>
    Phase_contents (FAPPLY (Tracker_phases trk) pid) SUBSET
    FDOM (Tracker_containers trk)''
```

we can deduce that

```
cid' IN FDOM (Tracker_containers trk)
```

Finally, the goal is solved using the assumption

```
''!pid.
    pid IN FDOM (Tracker_phases trk) ==>
    (!cid.
      cid IN Phase_contents (FAPPLY (Tracker_phases trk) pid) ==>
      cid IN FDOM (Tracker_containers trk) /\
      Container_material (FAPPLY (Tracker_containers trk) cid) IN
      Phase_expected_materials (FAPPLY (Tracker_phases trk) pid))''
```

Range or domain Consider the function

```
val Consistent_DEF = func 'Consistent(containers,phases) =
 !pid. pid IN FDOM phases ==>
       Phase_contents (FAPPLY phases pid) SUBSET FDOM containers';
```

from the tracker example. Now, consider the modified tracker example with the alternative definition of `Consistent`

```
val Consistent_DEF_2 = func 'Consistent(containers, phases) =
  !ph.
     inv_Phase ph ==>
     ph IN FRANGE phases ==>
     Phase_contents ph SUBSET FDOM containers';
```

where the quantification over the domain of the map `phases` is replaced by a quantification over its range.

We have applied the invariants tactic to each of the proof obligations, and none were proved valid. For example `tracker_PO14` resulted in one remaining subgoal

```
``Container_material
    (FAPPLY
       (DRESTRICT (Tracker_containers trk)
         (FDOM (Tracker_containers trk) DIFF {cid}))
      cid') IN
  Phase_expected_materials ph``
------------------------------------
    ``cid' IN Phase_contents ph``
    ``ph IN FRANGE (DRESTRICT (Tracker_phases trk) (\x. ~(x = source)))``
    ``Phase_expected_materials ph <> {}``
    ``CARD (Phase_contents ph) <= Phase_capacity ph``
    ``FINITE (Phase_expected_materials ph)``
    ``FINITE (Phase_contents ph)``
    ``cid IN Phase_contents (FAPPLY (Tracker_phases trk) source)``
    ``source IN FDOM (Tracker_phases trk)``
    ``!ph.
        FINITE (Phase_contents ph) /\
        FINITE (Phase_expected_materials ph) /\
        CARD (Phase_contents ph) <= Phase_capacity ph /\
        (Phase_expected_materials ph <> {}) ==>
        ph IN FRANGE (Tracker_phases trk) ==>
        (!cid.
          cid IN Phase_contents ph ==>
          cid IN FDOM (Tracker_containers trk) /\
          Container_material (FAPPLY (Tracker_containers trk) cid) IN
          Phase_expected_materials ph)``
    ``~(?p1 p2.
         p1 IN FDOM (Tracker_phases trk) /\
         p2 IN FDOM (Tracker_phases trk) /\
         (p1 <> p2) /\
         (Phase_contents (FAPPLY (Tracker_phases trk) p1) INTER
          Phase_contents (FAPPLY (Tracker_phases trk) p2) <>
          {}))``
    ``!ph.
        FINITE (Phase_contents ph) /\
        FINITE (Phase_expected_materials ph) /\
        CARD (Phase_contents ph) <= Phase_capacity ph /\
        (Phase_expected_materials ph <> {}) ==>
        ph IN FRANGE (Tracker_phases trk) ==>
        Phase_contents ph SUBSET FDOM (Tracker_containers trk)``
    ``!pid.
        pid IN FDOM (Tracker_phases trk) ==>
        FINITE (Phase_contents (FAPPLY (Tracker_phases trk) pid)) /\
        FINITE
          (Phase_expected_materials (FAPPLY (Tracker_phases trk) pid)) /\
        CARD (Phase_contents (FAPPLY (Tracker_phases trk) pid)) <=
        Phase_capacity (FAPPLY (Tracker_phases trk) pid) /\
        (Phase_expected_materials (FAPPLY (Tracker_phases trk) pid) <>
         {})``
```

The subgoal can be solved along the lines of the proof at the previous section.

Chapter 5

Implicitly defined functions

In this chapter, we report on one experiment that we did with the solving of proof obligations arising from implicitly defined functions, i.e., functions specified by pre- and post-conditions. Such proof obligations express that, for every element of the domain of the implicitly defined function, there exists an element satisfying the postcondition. The only such proof obligation occurring in the examples is

```
val alarm_PO4 =
  Term `!a per plant. inv_Plant(plant) ==>
      (pre_ExpertToPage(a,per,plant) ==>
      ?r. inv_Expert(r) /\ post_ExpertToPage(a,per,plant,r))`;
```

In general we expect this kind of proof obligations to be hard as they involve the construction of witnesses. However in some cases, they are rather straightforward because the witness is directly given by the context or the precondition. For alarm_PO4, we get the following subgoal after rewriting with definitions, splitting and an application of RES_TAC.

```
    ``?r.
       (Expert_quali r <> {}) /\
       r IN FAPPLY (Plant_schedule plant) per /\
       Alarm_quali a IN Expert_quali r``
  ------------------------------------
       ``a IN Plant_alarms plant``
       ``per IN FDOM (Plant_schedule plant)``
       ``!a.
          a IN Plant_alarms plant ==>
          (!per.
            per IN FDOM (Plant_schedule plant) ==>
            (?ex.
              ex IN FAPPLY (Plant_schedule plant) per /\
              Alarm_quali a IN Expert_quali ex))``
       ``ex IN FAPPLY (Plant_schedule plant) per``
       ``Alarm_quali a IN Expert_quali ex``
```

This is exactly the kind of goals that the tactic OUR_INV_EXISTS_TAC is aimed at and indeed when applied to the goal it returns the following subgoal

Example	PO	Duration (sec.)	vdm_split_tac	vdm_inv_exists_tac	vdm_contradict_tac	vdm_deduct_tac	vdm_simp_tac	vdm_res_tac	max_depth	backtracking	max_subgoals
Alarm	4	3.0	2	2	1	2	2	3	1	2	3

Table 5.1: Summary of experiments on satisfiability proof obligations.

```
``(Expert_quali ex <> {}) /\
  ex IN FAPPLY (Plant_schedule plant) per /\
  Alarm_quali a IN Expert_quali ex``
-------------------------------------
    ``a IN Plant_alarms plant``
    ``per IN FDOM (Plant_schedule plant)``
    ``!a.
        a IN Plant_alarms plant ==>
        (!per.
          per IN FDOM (Plant_schedule plant) ==>
          (?ex.
            ex IN FAPPLY (Plant_schedule plant) per /\
            Alarm_quali a IN Expert_quali ex))``
    ``ex IN FAPPLY (Plant_schedule plant) per``
    ``Alarm_quali a IN Expert_quali ex``
```

Table 4.2 summarizes the proof. The table should be read as explained for Table 4.1 in Section 4.2.

Chapter 6

Termination

In this chapter, we briefly consider proof obligations generated to ensure termination of recursive functions.

The VDM-SL examples in [5] contain one recursively defined function in the gateway example

```
val Gateway_DEF = rfunc `Gateway(ms,cat) =
    COND (ms = []) (mk_Ports([],[]))
                   (let rest_p = Gateway(TL ms,cat)
                    in
                      ProcessMessage(HD ms,cat,rest_p))`
     `measure \(x,y). LENGTH x`;
```

We use the `tfl` HOL98-theory [10, 12] to define functions. On top of the `Rfunction` of `tfl`, we have built a function `rfunc` which essential calls the function `Rfunction` which derives a termination condition and tries to solve it. If the `Rfunction` fails to solve the termination condition the `rfunc` makes an additional attempt. The definition is found in the file `vdm-func-def.sml` in [4] but it is rather ad hoc and will be changed eventually. We have chosen to supply a measure by hand at the moment but the function `function` of the `tfl` theory could just as well have extracted it for us in this case. The result of the definition above is

```
val Gateway_DEF =
   {induction =
      [oracles: MK_THM] [axioms: ] []
      |- !P.
           (!ms cat. (~(ms = []) ==> P (TL ms,cat)) ==> P (ms,cat)) ==>
           (!v v1. P (v,v1)),
    rules =
      [oracles: MK_THM] [axioms: ] []
      |- Gateway (ms,cat) =
         ((ms = [])
          => (mk_Ports ([],[]))
          | (let rest_p = Gateway (TL ms,cat)
             in
             ProcessMessage (HD ms,cat,rest_p))), tcs = []}
```

The fact that *tcs* is empty means that the termination of the functions was proved. Another, reason for chosing the `tfl` theory for the definition of functions is the `induction` entry above in which an induction principle for the function is returned. In the Chapter 7, the usefulness of the derived induction principle will be illustrated.

Chapter 7

General properties

So far, we have been concerned with the solving of proof obligations. In this chapter, we briefly report on some experiments with the proving of general properties or *validation conjectures* of specifications. The conjectures which we consider are taken from the gateway example and express properties about the gateway function

```
val CONJ_NoLost =
  Term `!ms cat.
    ((inv_Message_seq ms /\ inv_Category cat) ==>
     (let port = Gateway(ms,cat) in
        let hi = Ports_high port and
            lo = Ports_low port
        in (ELEMS ms = (ELEMS hi UNION ELEMS lo)) /\
           LENGTH ms <= LENGTH hi + LENGTH lo))`;

val CONJ_NoBothPorts =
  Term `!ms cat.
     ((inv_Message_seq ms /\ inv_Category cat) ==>
      (let port = Gateway(ms,cat) in
         let hi = Ports_high port and
             lo = Ports_low port
         in (ELEMS hi INTER ELEMS lo) = {}))`;

val CONJ_NoHighOnLow =
  Term `!ms cat.
     ((inv_Message_seq ms /\ inv_Category cat) ==>
      (let port = Gateway(ms,cat) in
         let hi = Ports_high port and
             lo = Ports_low port
         in !m. (m IN (ELEMS lo)) ==> ~(Classify(m,cat) = HI)))`;

val CONJ_SameFunction =
  Term `!ms cat.
    (inv_Message_seq ms) ==> (inv_Category cat) ==>
    ((Gateway(ms,cat))=(Gateway2(ms,cat)))`;
```

We made some head-on attempts to prove the properties by induction using the induction principle derived from the definition of the recursive function gateway,

see Chapter 6, and the support tactic PROGRAM_TAC of the tfl theory.

We begin with the first conjecture, *CONJ_NoLost*, stating that no message is lost in the gateway. Using the following tactic:

```
fun VDM_INDUCTION_TAC (facts:{induction : Thm.thm,
                              rules : Thm.thm,
                              tcs : Term.term list})=
  (PROGRAM_TAC {induction = #induction(facts),
                rules = #rules(facts)}) THEN
  (POP_ASSUM_LIST (MAP_EVERY MP_TAC)) THEN
  VDM_RW_DEF_TAC THEN
  (VDM_REC_INV_TAC 0);
```

based on the PROGRAM_TAC of tfl and the VDM_INV_TAC from above, the conjecture is shown in a few seconds. Note that the PROGRAM_TAC takes the induction derived when defining the Gateway function in HOL98 as an argument.

Trying to prove the second conjecture CONJ_NoBothPorts in the same naive way fails.

A closer look at the subgoals below tells us that in fact CONJ_NoBothPorts is not strong enough to be used as induction hypothesis.

```
``F``
------------------------------------
    ``!hi i j.
        hi IN cat /\ i IN INDS (HD ms) /\ j IN INDS (HD ms) ==>
        (hi <> SUBSEQ (HD ms) i j)``
    ``HI <> LO``
    ``ELEMS (Ports_high (Gateway (TL ms,cat))) INTER
      ELEMS (Ports_low (Gateway (TL ms,cat))) =
      {}``
    ``!m. m IN ELEMS (TL ms) \/ (m = HD ms) ==> LENGTH m <= 100``
    ``!m. m IN ELEMS (TL ms) \/ (m = HD ms) ==> ([] <> m)``
    ``!s. s IN cat ==> ([] <> s)``
    ``[] <> ms``
    ``HD ms IN ELEMS (Ports_high (Gateway (TL ms,cat)))``

``F``
------------------------------------
    ``SUBSEQ (HD ms) i j IN cat``
    ``i IN INDS (HD ms)``
    ``j IN INDS (HD ms)``
    ``hi = SUBSEQ (HD ms) i j``
    ``T``
    ``ELEMS (Ports_high (Gateway (TL ms,cat))) INTER
      ELEMS (Ports_low (Gateway (TL ms,cat))) =
      {}``
    ``!m. m IN ELEMS (TL ms) \/ (m = HD ms) ==> LENGTH m <= 100``
    ``!m. m IN ELEMS (TL ms) \/ (m = HD ms) ==> ([] <> m)``
    ``!s. s IN cat ==> ([] <> s)``
    ``[] <> ms``
    ``HD ms IN ELEMS (Ports_low (Gateway (TL ms,cat)))``
```

In other words, the subgoals ask to show that whenever a message m is correctly added to the high or low port then m is not already on the other port, a property that is not guaranteed by the induction hypothesis. Hence, a stronger hypothesis

Conjecture	Duration (sec.)	vdm_split_tac	vdm_inv_exists_tac	vdm_contradict_tac	vdm_deduct_tac	vdm_simp_tac	vdm_res_tac	max_depth	backtracking	max_subgoals
CONJ_NoLost	7.8	6	4	4	6	6	6	1	6	2
CONJ_NoBothPorts	–	102	106	106	102	102	153	6	102	2
CONJ_NoHighOnLow	1355.4	67	83	83	67	67	121	4	67	8
CONJ_SameFunction	61.2	20	22	22	20	20	20	5	20	2

Table 7.1: Summary of experiments with subtype checking proof obligations.

is needed. Note that we produced the subgoals afterwards – by hand! How does a strategy performing proof search produce "good" subgoals in case of failure?

Also, the third and fourth conjectures are not proved by the strategy. Instead, we attempt the following slightly changed recursive tactic

```
fun VDM_REC_INV_TAC_2 (depth:int):tactic = fn g => (
  let val _ = max_depth := (if depth > (!max_depth) then depth
                                                      else (!max_depth))
  in
    VDM_SPLIT_TAC THEN
    ((SIMP_TAC vdm_ss vdm_rewrites) THEN
    FIRST
      [OUR_ASM_ACCEPT_TAC,
       COND_CASES_TAC THEN (VDM_REC_INV_TAC (depth+1)),
       (VDM_INV_EXISTS_TAC (VDM_REC_INV_TAC (depth+1))),
       VDM_CONTRADICT_TAC,
       (VDM_DEDUCT_TAC (VDM_REC_INV_TAC (depth+1))),
       (VDM_SIMP_TAC (VDM_REC_INV_TAC (depth+1))),
       (VDM_RES_TAC (VDM_REC_INV_TAC (depth+1))),
       (BACKTRACKING_TAC depth)
      ]
    )
  end
) g;
```

which is built on top of VDM_REC_INV_TAC extending it by letting the component tactic VDM_RES_TAC call VDM_REC_INV_TAC.
Table 7.1 summarizes the results obtained by applying the induction tactic above with VDM_REC_INV_TAC replaced by VDM_REC_INV_TAC_2 to each of the conjectures. The table should be read as explained for Table 4.1 in Section 4.2. The grey shaded box marks that a the strategy failed. All conjectures but CONJ_NoBothPorts were proved valid. When applying the tactic to CONJ_NoBothPorts, it looped several times while simplifying and eventually crashed while simplifying.

Chapter 8

Conclusions

In this report we have discussed the verification of VDM-SL proof obligations using HOL98. Proof obligations for domain checking turned out to be particularly easy to prove. All but one of these were solved using the propositional tautology checker in HOL. Proof obligations for subtype checking arise due to the use of invariants in type definitions and are typically more difficult to verify. We presented an adhoc recursive strategy which applies successively more powerful proof steps in order to solve a proof obligation. The strategy proves each of the considered proof obligations within minutes.

The experiments have been both encouraging and discouraging. It is encouraging to see that so many proof obligations can be solved using simple and efficient techniques like tautology checking. This can be a valuable approach to reducing the number of proof obligations presented to a user of the VDM tools. It is also encouraging to see that we are able to build a powerful strategy to solve even more proof obligations automatically. The strategy is based mainly on general proof steps like simplification and top-down deduction where the conclusion of a goal is reduced to simplier subgoals. HOL resolution is applied in a limited way when nothing else seems to work.

However, we are not confident in the robustness of this strategy for two reasons. First of all, it was developed in a demand-driven way while verifying the example proof obligations, so it is difficult to predict how well it will work in general. We have seen that even smaller changes can affect the success of the strategy, because the order in which proof steps are performed is important and the strategy employs heuristic approaches, for example, to invent witnesses for existential goals. Secondly, the proof steps of the strategy are mostly based on the application of theorems, and so the quality of our theorem database is absolutely essential. Presently the theorems and simplification sets in the database are adhoc, but in the future we hope to make a systematical selection of theorems from relevant theories and organize them in suitable simplification sets. However, it is impossible to say how powerful this will make the strategy in practice. More experiments are needed to show this.

Another discouraging aspect is the size and number of subgoals that arise in

proofs. One proof obligation can split into hundreds of subgoals and one subgoal can be many pages long, as illustrated in Figure 4.1 and Figure 4.2. These problems are partly related to the brute-force approach of the proof strategy, which expands all VDM-SL definitions to constructs in pure HOL. If expansions are made in a more controlled way, then the subgoals and proofs will be smaller and easier to present to a user. However, the trade-off is that this approach requires interaction with a user in order to obtain suitable lemmas for VDM-level concepts and constructs. This shows a difference between automatic and interactive theorem proving; the latter is structured more like a paper-and-pencil proof. Moreover, the present version of the automatic strategy could be made more sophisticated and intelligent by programming it to analyse goals before searching for proofs. Perhaps, this could be supported by some kind of tagging produced by the proof obligation generator. For example, the proof obligation generator could tag easy goals in a certain way.

It is often difficult for an automatic strategy to produce good output which is easy to read and comprehend, when it does not work. We have thought about different approaches to simplifying unproven subgoals of our proof strategy, e.g. by reversing expansions (if possible) or tagging assumptions with information on how they were deduced to simplify presentation, but none of these work well. So our present philosophy is to leave those subgoals unchanged and complex, though translations are naturally reversed to the VDM level, and focusing instead on supporting the interactive verification of difficult proof obligations well, when the automatic strategy fails. In interactive proving, it is up to the user to control the expansions and other deductions, but even then we believe that it is important to have flexible ways of hiding and viewing information like assumptions or parts of assumptions.

We have experimented with most built-in automated proof procedures of HOL98. However, it is quite difficult to determine what they can do for us, and what they cannot do. This could be solved partly by better documentation including user guidelines.

Profiling support would help in order to ease experimental investigations using different tactics and proof strategies. By profiling we understand the collection of statistical information such as time and space required to do a proof, but it should also include more detailed information about the structure of the proof, the number of subgoals, which tactics solved which subgoals, which tactics failed to solve which subgoals, how much time was spend by individual tactics such as the simplifier, and so forth. In this report, we have presented some primitive profiling information collected by dummy tactics, which is not precise enough to be really useful.

In the experiments conducted so far, we have naturally not been able to work with not-yet-integrated proof procedures. We have, however, initiated collaborative work with Prover Technology AB on using their decision procedure for propositional logic, called Prover, as well as their upcoming decision procedure for first-order many-sorted logic. For this collaboration, we have translated the alarm example in [5] into an intermediate language without type invariants, record types, sequence types, and mappings (see Appendix A), which will form

the basis for a translation into the logics supported by the Prover tools. A completely manual translation to propositional logic is not feasible due to an explosion in the number of variables needed to represent VDM-SL specifications in such a format. This explosion happens even though infinite domains like natural numbers and the set type have been restricted to finite domains.

The most surprising lessons we have learnt from our experiments are:

- The usefulness of the simplifier, though it needs instantiation with appropriate sets of rewriting theorems and sometimes does not terminate or does not terminate quickly enough.
- The size and complexity of subgoals, especially compared with the size of the examples.
- The power of mixing top-down deduction and simplification while limiting the use of HOL resolution, which should be used particularly carefully together with simplification.
- The difficulty of understanding and applying some of the incorporated decision procedures of HOL.

Appendix A

Appendix

```
(* The Alarm example in many sorted FOL *)

Sorts
  Alarm_alarmtext_inds = nat1;

  Qualification = Elec | Mech | Bio | Chem;
  ExpertId = token; (* Some infinite set of elements - supporting equality *)
  Period = token;   (* Some infinite set of elements - supporting equality *)

  Expert = ExpertId * set of Qualification;
  Schedule = set of (Period * set of Expert);
  Alarm = (set of (Alarm_alarmtext_inds * char)) * Qualification;

  Plant = Schedule * set of Alarm;

(* functions supported (by YOU):

  x projections on product sorts
    prj1: A * B -> A  and   prj2: A * B -> B

  x membership relation for set of sorts.

  x choose operator on set sorts (set of A):
    choosing an element in A satifying a predicate P:A -> bool:
    choose: set of A * (A -> bool) -> A

  x Encoding / axiomatisation of the empty set
    EmptySet

*)

(* functions definitions *)

  (* Auxillary functions *)

  Map_Alarm_alarmtext(m) =
    forall x,y: Alarm_alarmtext_inds * char.
      (in(x,m) /\ in(y,m)) =>
        prj1(x)=prj1(y) => prj2(x)=prj2(y);
```

```
  In_Dom_Alarm_alarmtext(m) =
    exists elm:Alarm_alarmtext_inds * char.
      in(elm,m) /\ prj1(elm) = per;

  Is_Seq(sa) = Is_Map_Alarm_alarmtext(sa) and
    exists m: Alarm_alarmtext_inds.
      forall n: Alarm_alarmtext_inds.
        (n <= m) => In_Dom_Alarm_alarmtext(n,sa) /\
        (n > m) => ~In_Dom_Alarm_alarmtext(n,sa);

  Map_Schedule(sch) =
    forall x,y: Period * set of Expert.
      (in(x,sch) /\ in(y,sch)) =>
        prj1(x)=prj1(y) => prj2(x)=prj2(y);

(* The following: Not quite right - any ideas for
   encoding / axiomatising
   the choice of an element in a set - what will you provide??
*)

  Lookup_Schedule(sch,per) =
    Prj2_Schedule(choose(sch,
      lambda p : Period * set of Expert &
             in(p,sch) /\ (Prj1_Schedule(p) = per)));

  In_Dom_Schedule(per,sch) = exists elm:Period * set of Expert.
                             in(elm,sch) /\ prj1(elm) = per;

  In_Rng_Schedule(exs,sch) = exists elm:Period * set of Expert.
                             in(elm,sch) /\ prj2(elm) = exs;

  Prj1_Schedule(p) = prj1(p);

  Prj2_Schedule(p) = prj2(p);


  (* Structured names for projections *)

  Expert_expertid (e) = prj1(e);
  Expert_quali (e) = prj2(e);

  alarm_alarmtext(a) = prj1(a);
  alarm_quali(a) = prj2(a);

  plant_schedule(p) = prj1(p);
  plant_alarms(p) = prj2(p);

  (* Invariants *)

  inv_Expert(ex) = Expert_quali(ex) <> EmptySet;

  inv_Schedule (sch) =
    Map_Schedule(sch) /\
    (forall exs: set of Expert. In_Rng_Schedule(exs,sch) =>
      forall ex: Expert. in(ex,exs) => inv_Expert(ex)) /\
    (forall exs: set of Expert.
```

```
      In_Rng_Schedule(exs,sch) =>
      ( (exs <> EmptySet) /\
      forall ex1 ex2: Expert. (in(ex1,exs) /\  in(ex1,exs)) =>
        ((ex1 <> ex2) =>
         (Expert_expertid(ex1) <> Expert_expertid(ex2)))));

  inv_Plant(plant) =
    inv_Alarm(plant_alarms(a)) /\
    forall a: Alarm.  inv_Alarm(a) => in(a,plant_alarms(plant)) =>
      forall per:Period. In_Dom_Schedule(per,plant_schedule(plant)) =>
        QualificationOK(Lookup_Schedule(plant_schedule(plant),per),
                        alarm_quali(a));

  inv_Alarm(a) = Is_Seq(alarm_alarmtext(a));

  (* user defined functions *)

  QualificationOK (exs,reqquali) =
    exists ex: Expert.  inv_Expert(ex) =>
      in(ex,exs) => in(reqquali,Expert_quali(ex));

  pre_ExpertToPage(a,per,plant) =
        In_Dom_Schedule(per,plant_schedule(plant)) /\
        in(a,plant_alarms(plant));

  post_ExpertToPage(a,per,plant,r) =
      in(r,Lookup_Schedule(plant_schedule(plant),per)) /\
      in(alarm_quali(a),Expert_quali(r));


(* Questions *)
  (* Proof obligations *)
  PO1 = forall a:Alarms, per:Period, schedule:Schedule.
        inv_Alarms(a) => inv_Schedule(schedule) =>
        (in(a,alarms) /\  In_Dom_Schedule(per,schedule)) =>
         In_Dom_Schedule(per,schedule))

  PO2 = forall per:Period, plant:Plant.
        inv_Plant(plant) =>
        (In_Dom_Schedule(per,plant_schedule(plant)) =>
         In_Dom_Schedule(per,plant_schedule(plant)))

  PO3 = forall a:Alarm, per: Period, plant:Plant.
        inv_Alarm(a) => inv_Plant(plant) =>
          pre_ExpertToPage(a,per,plant) =>
            In_Dom_Schedule(per,plant_schedule(plant))

  PO4 = forall a:Alarms, per:Period, plant:Plant.
        inv_Alarm(a) => inv_Plant(plant) =>
        (pre_ExpertToPage(a,per,plant) =>
        exists r:Expert. inv_Expert(r) /\ post_ExpertToPage(a,per,plant,r));
```