ANR-08-DEFI-005
DECERT

Task 1: Report on Requirement Analysis

The Decert Consortium

July 1, 2009

Contents

Chapter 1

Introduction

This document reports on the results obtained in *Task 1* of *Work Package 1*. The overall aim of this task was to gather in a central place the detailed requirements from partners on the expected results of the project. The requirements analysed during this task have been organised along the three axes of the project: expressiveness, efficiency and proof witnesses

As concerns expressiveness, the requirements take the form of typical problems (mathematical lemmas) which are often encountered in practice and not yet well supported by proving tools. These lemmas are organized in a taxonomy. This taxonomy has been defined along the nature of the arguments that would allow to discharge them (linear arithmetics, simple set-theory, list theory, ...). A special attention has been devoted to the cases where only a combination of arguments of different natures would succeed. As far as possible, these lemmas are expressed directly in the SMT-LIB format.

As concerns efficiency, the requirements take the form of expected time and space necessary for solving some typical problems. Finally, for proof witnesses, the requirements are both qualitative (on the format used) and quantitative (on the size of the proof witnesses, space and time necessary for checking them).

Chapter 2

Expressiveness

2.1 Taxonomy

2.1.1 List of theories

One or many *theories* can be associated to a lemma. These theories characterize the nature of the arguments that would allow to discharge the lemma. The list of theories are :

boolean: sort BOOL containing exactly two members: constants FALSE and TRUE

arrays: functional arrays without extensionality

basic set: basic sets with extensionality. No sets of sets allowed. Supports classical operations: membership test, union, intersection, set difference.

basic relation: basic relations with extensionality. Relations are represented as sets of pairs. In addition to classical set operations, also support domain, range, domain restriction, functional overriding, etc.

full set theory: unrestricted multi-sorted set theory

integer: sort $\mathbb{Z}$ and constants for integer literals

linear order int: classical linear order on the integers ($\leq$)

linear arith: linear arithmetic on the integers (adds support for addition and subtraction)

nonlinear arith: nonlinear arithmetic on the integers (adds support for multiplication)

full arith: full arithmetic on the integers (adds support for exponentiation)

linear order real: classical linear order on the reals ($\leq$)

linear real arith: linear arithmetic on the reals

nonlinear real arith: nonlinear arithmetic on the reals

full real arith: full arithmetic on the reals

In the above list, a theory may be included into another one. In other words, given two theories *th1* and *th2*, the set of problems that can be expressed in *th1* may be a strict subset of that of *th2*. For example, the lemma

$$x \in \mathbb{Z} \wedge d \in \mathbb{Z} \wedge x \geq d \Rightarrow x + 1 > d$$

is part of *linear arith*, but the comparison operators $\geq$ and $>$ are also part of *linear order int*. Symbolically, we have:

$$\textit{basic set} \subset \textit{basic relation} \subset \textit{full set theory}$$

and

$$\textit{integer} \subset \textit{linear order int} \subset \textit{linear arith} \subset \textit{nonlinear arith} \subset \textit{full arith}$$

2.1.2 Combination of theories

In most cases, a single theory is not powerful enough to prove a lemma. This is because the proof needs to resort to a combination of different arguments, each belonging to a different theory.

For instance, to prove the lemma

$$p \leq q \;\Rightarrow\; f[p \,..\, q] = f[p \,..\, (q-1)] \cup f[\{q\}]$$

a combination of two theories is needed:

- *basic relation* allows to reason about relational images and to use the theorem

$$f[A \cup B] = f[a] \cup f[B]$$

- *linear order int* is needed to reason about integer interval and provides theorem

$$p \leq q \;\Rightarrow\; p \,..\, q = p \,..\, (q-1) \cup \{q\}$$

2.2 Rodin Benchmarks

2.2.1 Derivation of lemmas

These lemmas have been extracted from various projects developed with the event-B notation. These projects come both from Academia (for instance, all the examples from Jean-Raymond Abrial's coming book) and Industry (case studies as part of the IST European project Deploy).

All lemmas are stored in XML files validated against a DTD available in the same directory (see file `lemmas.dtd`).

2.2.2 Volumetry

The table 2.1 gives the theory combinations encountered in extracted lemmas. These combinations are ordered in function of their occurrence numbers. The table 2.2 gives the occurrences for each theory.

Theory Combination	Nb Lemmas
linear arith; *basic relation*	67
linear arith	60
linear order int; *full set theory*	46
full set theory	46
linear order int; *basic relation*	42
linear arith; *full set theory*	29
linear order int	20
nonlinear arith	19
linear order int; *basic set*	13
linear arith; *basic set*	11
basic relation	10
integer; *full set theory*	5
integer; *basic relation*	4
full set theory; *nonlinear arith*	4
basic set	3
integer	1
Total	380

Table 2.1: Number of lemmas per theory combination.

Theories	Nb Lemmas
linear arith	168
full set theory	130
basic relation	123
linear order int	121
basic set	27
nonlinear arith	23
integer	10

Table 2.2: Number of Lemmas per Theory.

- Lemmas involving integer manipulation are not well supported by the current *Rodin* platform (theories *linear order int*, *linear arith* and *nonlinear arith*).
- When a lemma involves manipulation of complicated structures (for example sets of functions) provers in the *Rodin* platform are often disturbed. In fact, many lemmas can be discharged by replacing complicated terms by simpler ones (command *ae*). This explains why there are many lemmas of *full set theory* which are not discharged by current *Rodin* provers.

2.2.3 Translation to SMT-LIB format

In order to profit *SMT-LIB* provers, a lemma needs to be rewritten in *SMT-LIB* format. Currently, only lemmas of the following theory combinations have been translated into the *SMT-LIB* format:

- *linear arith*
- *linear order int*

The translator is available open-source on SourceForge (at the same place as the sources of the Rodin platform). This translator constitutes the basis for the later integration of SMT solvers in the Rodin platform that will be carried on in Task 7.

2.3 Why Benchmarks

The Why platform [6] is made of three components:

- a programming language;
- a first order polymorphic logic;
- a verification condition generator (VCG).

The VCG translates a program annotated with its specification into a Why formula. The input program may be written in the C language or in Java. For that purpose, it is processed into the Why language and then handled by the Why VCG. The original program meets its specification whenever the obtained formula is valid. This approach is summarized in Figure 2.1.

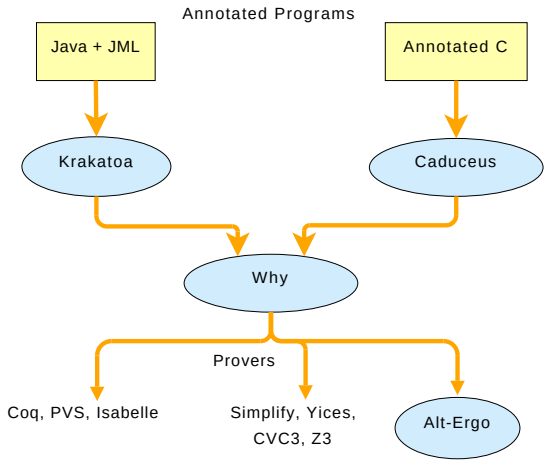


Figure 2.1: The Why tool chain

The formulas collected here come from annotated Java programs or native Why programs. The described processing of C programs is now obsolete and replaced by the Frama-C approach which is described in the Frama-C section 2.4.

The Why formulas are dispatched in several directories named after their input program. Each formula comes in two formats: the Why logic syntax and the SMT-Lib syntax [17]. Among the proof obligations, we have selected only those which are proved in less than 10 seconds by at most one among the following SMT provers: Alt-Ergo, CVC3, Simplify, Yices and Z3.

The directories are listed below:

- algo64
- algo65
- arith
- bresenham
- dijkstra
- fib
- find
- heapsort
- krakatoa
- maximumsort
- mergesort
- min_max

- peano
- power
- queens
- quicksort
- selection
- sqrt
- sqrt_dico
- sum

The krakatoa directory is also divided in seven sub-directories (Arrays, Fibonacci, FlagStatic, Gcd, Muller, Purse, Sort).

For example, the formulas selected in the quicksort directory come from the following annotated quicksort program written in the Why language. Annotations are between brackets and expressed variants and invariants (pre- and post-conditions) of functions.

```
(* The first part of the program that re-arrange elements in the array
   and returns the position of the "partition" is defined in the module
   [Partition_prog]. *)

(* The recursive part of the quicksort algorithm:
   a recursive function to sort between [l] and [r] *)

let quick_rec =
  let rec quick_rec (t:int array)(l,r:int) : unit
  { variant r-l } =
    { 0 <= l and r < array_length(t) (*as Pre*) }
    (if l < r then
       let p = (partition t l r) in
       begin
         (quick_rec t l (p-1));
 (quick_rec t (p+1) r)
       end)
    { sorted_array(t, l, r) and permut(t, t@, l, r) }

(* At last, the main program, which is just a call to [quick_rec] *)

let quicksort =
  fun (t:int array) ->
    {}
    (quick_rec t 0 ((array_length_ t)-1))
    { sorted_array(t, 0, array_length(t)-1) and permutation(t, t@) }
```

The first selected proof obligation (`po71.why`) corresponds to the left part of the post-condition of the function `quick_rec`: `sorted_array(t, l, r)`:

```
goal quick_rec_po_8:
  forall l:int.
  forall r:int.
  forall t:int farray.
  ((0 <= l) and (r < array_length(t))) ->
  (l < r) ->
  (((0 <= l) and (l < r)) and (r < array_length(t))) ->
  forall result:int.
  forall t0:int farray.
  (((l <= result) and (result <= r)) and
```

```
 (partition_p(t0, l, r, result) and permut(t0, t, l, r))) ->
((0 <= (r - l)) and (((result - 1) - l) < (r - l))) ->
((0 <= l) and ((result - 1) < array_length(t0))) ->
forall t1:int farray.
(sorted_array(t1, l, (result - 1)) and permut(t1, t0, l, (result - 1))) ->
((0 <= (r - l)) and ((r - (result + 1)) < (r - l))) ->
((0 <= (result + 1)) and (r < array_length(t1))) ->
forall t2:int farray.
(sorted_array(t2, (result + 1), r) and permut(t2, t1, (result + 1), r)) ->
sorted_array(t2, l, r)
```

So far, we have collected 125 proof obligations.

2.4 Frama-C Benchmarks

The benchmarks from CEA come from the program verification technique called weakest pre-condition calculus [7, 18] which takes as input a function f and a specification of what f should do and generates a set of proof obligations. If each of these proof obligations holds, f is conforming to its specification.

The CAVEAT [16] and Frama-C [19, 14] tools apply these techniques to C code, in particular programs coming from critical embedded code. In this context, the main characteristics of the formulas given to the prover can be summarized as follows:

- large number of formulas: the number of proof obligations grows with the size of the function under analysis. In particular, there are usually numerous safety properties (pointer accesses are always valid, no division by 0, no overflow,...) that need to be established.
- size of the formula: Informally, a proof obligation can be divided into a goal (the desired property) and a set of hypotheses which is a summary of all the computations that have been made along an execution path. Again, this set tends to become quite large as the analyzed code grows.
- theories used: properties involves mainly equality and uninterpreted functions, array accesses and integer arithmetic (mostly linear, but non-linear computations occur on a regular basis). C programs use bounded integer but this is taken care of at the tools' level by generating proof obligations stating that every arithmetic operation stays within the bounds of the corresponding integer type, so that the result is the same as for unbounded integers. Some programs rely on the modulo arithmetic for unsigned integers, thus bringing the need for modulo arithmetic within the provers, but this is very rare. Real and floating-point arithmetic are also of interest, but we primarily rely on specialized tools (especially Gappa) for such properties, so that it might fell outside of the scope of this project.

2.4.1 Efficiency requirements

Given the number of proof obligations that can be generated, efficiency of the provers is very important. Moreover, the provers must be able to achieve reasonable performance without relying on explicit triggers or other external configuration. While it is admissible to tweak the prover to help it verify a complex functional property, "standard" safety properties should not require any intervention at all. Memory consumption is less of an issue.

It is also important that the provers be able to cope with large numbers of hypotheses, which might for a substantial part be unrelated to the particular goal at hand. It has been shown [8, 12] that it is sometimes possible to remove some hypotheses that are unrelated to the goal and that it results in a significant improvement in what some automated provers are able to handle. An integration of such a "pre-processing" step into the provers might thus be useful.

More specifically, some previous experiments with alt-ergo have shown some specific areas where improvements would be the most useful. This includes in particular the following:

- heavily quantified formulas and formulas involving a lot of constant parameters.
- proof by cases, where distinguishing several states (*e.g.* $x = 0$, $x = 1$, $x = 2$ and $x \geq 3$) would facilitate the proof search.
- sensitivity to the syntactic form of the property (*i.e* two logically equivalent formulas where one is easily proved and not the other).
- non-linear arithmetic operations. DECERT kick-off has already shown that support for a limited subset of non-linear integer arithmetic is doable, but a few extensions to this subset might be interesting.

2.4.2 Benchmark summary

The examples provided in the `FramaC` directory are proof obligations generated by Caveat and two plugins of Frama-C using different memory models for deductive verification, which at least one prover among Z3, Simplify and alt-ergo has found valid, while one or two others failed to find a proof.

Some of the proof obligations stem from the regression test base of the tools and are meant to explore various tricky points (at least from Hoare logic's point of view) of C semantics while the others are representative of proof obligations arising from "real" C code. In term of the taxonomy of section 2.1, these formulas use mainly *boolean*; *linear arith*, with few problems in *boolean*; *nonlinear arith*. Each proof obligation comes as a why goal in `file.why` and a smtlib goal in `file_why.smt`.

More specifically, the test base is structured as follows:

- `caveat` directory contains proof obligations generated by Caveat from embedded code (267 test cases).
- `cert` directory contains proof obligations generated by the Jessie plugin of Frama-C for the verification of the absence of run-time error in CERT's managed string library [3], which has been studied by Yannick Moy during his PhD [14] (770 test cases).
- `frama-c-regtests` contains examples from Frama-C's test base. This is where some non-linear formulas occur (100 test cases).
- `temporal` contains proof obligations from temporal logic which assess that an automaton stays in a safe state. The formulas are much larger than in the other repositories (730 test cases).

2.5 veriT Benchmarks

2.5.1 Derivation of lemmas

The veriT solver is used for several verification tasks, and notably as a back-end of a platform to prove distributed lock-free data-structures. Non-blocking implementations of data-structures have been proposed as an alternative to distributed data-structures wrapped inside mutexes since at least the early 1990s and have received new attention with the advent of highly concurrent processor and system architectures. The underlying algorithms are based on atomic instructions offered by processors that permit optimistic concurrency: processes execute as if they had exclusive access and react appropriately when concurrent access is detected. Compare-And-Swap, Load-Link/Store-Conditional, and Test-And-Set are examples of such processor instructions. The most widely accepted correctness property that non-blocking algorithms should guarantee is linearizability: the externally observable behavior corresponds to some interleaving of atomic high-level operations. From the user's point of view, the implementation can therefore be understood in terms of a sequential specification where each operation is described in isolation. This property is proved using refinement, rely-guarantee invariants, and inductive invariants.

This set of benchmarks will also be enriched in the future by proof obligations originating from the TLA+ [9] prover. These formulas are the verification conditions necessary to prove the correctness of distributed algorithms like mutual exclusion, leader election,... The TLA+ prover is skeptical towards external provers, it will thus be required to provide checkable proofs for every validated verification condition.

2.5.2 Volumetry

The theories of the generated proof obligations depend on the data-structures; the verification conditions we propose are not very challenging in the theory they are written in, since they are simply quantified first-order logic formulas with only uninterpreted symbols. We provide 7 of them, in the SMT-lib format. The main challenge is to find the right instances of the quantifiers in these formulas, and to be able to provide a proof. Future proof obligations from the same application may contain some symbols from other theories, and in particular from linear integer arithmetic and arrays.

The benchmarks coming from the TLA+ prover are more challenging with respect to the theory. This theory will be some rich combination of the theories mentioned in the above taxonomy, including heavy use of sets and relations, and arithmetics. However the quantifier structure of the formulas should be simple, and their size much smaller.

2.6 Java PCC Benchmarks

Our benchmarks are proof obligations needed to ascertain the results of static analyses for Java byte-code programs. For the purpose of this project, the static analyses we consider are array-bound-checking and points-to analyses. The importance of these analyses is well-established and state-of-the-art works in the field propose as benchmarks a set of byte-code Java programs. We will evaluate our work against these very benchmarks.

For static analysis, speed and precision are the main criteria. Soundness is also of prominent importance but is usually almost impossible to assess. The context of DECERT will offer the opportunity for assessing the trustworthiness of the analyses because our analysis results will be validated by external proof-generating SMT provers. This specific aspect and the associated challenges will be further developed in Chapter 4.

For array-bound-checking, micro-benchmarks are sorting algorithms such as Bubblesort and Quicksort. The major benchmarks are the SPECjvm'98 suite [1], the Scimark 2.0 suite [2] and the Java Grande Forum Benchmark Suite [3]. These programs have been used for evaluating both static and dynamic analyses for eliminating array-bound checks [2, 15, 21, 11].

In 2008, Lhotak and Hendren [10] have published a comprehensive evaluation of context-sensitive points-to analyses for Java. Their benchmarks subsume previous studies. They are the JOlden benchmark suite[4], the SpecJVM 98 benchmark suite, the DaCapo benchmark suite [5], the Ashes benchmark suite [6] and the Polyglot Java front-end [7].

2.6.1 Verification conditions for static analyses

There is a gap between a set of benchmark programs for static analyses and SMT provers. In the following, we briefly explain how SMT verification conditions can be derived from a static analysis result.

Static analyses are designed following the methodology of abstract interpretation. The program concrete semantics is abstracted over an abstract lattice $(D, \sqsubseteq, \sqcup, \bot)$. The abstract program takes the form of a system of fixpoint equations to be solved by chaotic fixpoint iteration [4].

$$\begin{array}{lcl} x_1 & = & f_1(x_1, \ldots, x_n) \\ \ldots & & \\ x_n & = & f_n(x_1, \ldots, x_n) \end{array}$$

Typically, each x_i represents an invariant attached to a particular program point and the f_is are abstract transfer function that model the semantics of instructions at the abstract level.

Verifying the result of static analyses consists in providing logic counterparts to the f_i and generate abstract verification conditions of the form $f_i(x_1, \ldots, x_n) \sqsubseteq x_i$ to be discharged by decision procedures.

2.6.2 Non-linear arithmetic

Array-bound checking requires numeric reasoning that is embedded into numeric abstract domains such as convex polyhedra [5] and octagons [13]. These domains have a direct logic counterpart linear arithmetic (resp. difference constraint logic) capable of deciding the ordering $\sqsubseteq$ of the abstract domain. Yet, if the invariants are linear, the proof obligations do not fall into the fragment of linear arithmetic because the abstract transfer functions do perform non-linear reasoning – in a limited way. To discharge our proof obligations, our requirements over the expressiveness of the arithmetic decision procedure are:

- handling of Java arithmetic (32 and 64 bits) including addition, multiplication, division, modulo, conversions between numeric types, masks and shift operations;
- a restricted form of non-linear reasoning mimicking the inferences done by the abstract transfer functions.

Discharging non-linear proof obligations is interesting in its own right. In theory, the problem has an awful complexity and becomes undecidable for integers. To get a glimpse of the difficulty encountered

[1] http://www.spec.org/osg/jvm98/
[2] http://math.nist.gov/scimark2/
[3] http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html
[4] ftp://ftp.cs.umass.edu/pub/o8l/benchmarks/jolden.tar.gz
[5] http://www-ali.cs.umass.edu/DaCapo/gcbm.html
[6] http://www.sable.mcgill.ca/ashes
[7] http://www.cs.cornell.edu/projects/polyglot/

in practise we have discharged manually verification conditions for a number of programs taken from the repository of Rodriguez Carbonell [8]. For the programs `cohencu` , `cohendiv`, `dijkstra`, `divbin`, `fermat`, `prodbin` and `sqrt`, we provide the why programs and the associated Coq proofs. It appears that most of the proofs do not fall into decidable fragments. The most challenging require the power function and the non-linear proofs are only valid over the integers.

2.6.3 Verification conditions for points-to analyses

State-of-the-art points-to analyses are leveraging BDD technologies in order to compute efficiently the points-to graph of programs [15, 20, 1]. The BDD operations needed to compute the analysis result are union, intersection, and renaming. The benchmarks and analyser for Berndl *et al.,* [1] work is available [9]. From their rough data, we are working at generating SMT verification conditions. Because of the size of the BDDs, generating the verification conditions itself is challenging. Eventually, the logic we target for our verification conditions is QBF – the logic of quantified boolean formulae. This is needed in order to generate verification conditions that model the renaming of BDD variables. For the time being, we are experimenting with purely boolean verification conditions but their size is prohibitively big. For instance for the `compress` program of the SPECJVM the textual representation of the biggest proof obligation takes up-to 94 Megabytes.

[8] `http://www.lsi.upc.edu/~erodri/webpage/polynomial_invariants/list.html`
[9] `http://www.sable.mcgill.ca/bdd/`

2.7 Coq Benchmarks

Coq benchmarks belong to two categories: boolean and non-linear arithmetic.

2.7.1 Boolean

These benchmarks are standard SAT problems. Typical certificates for SAT problems are resolution proofs. There is no correlation between the size of the statement of problem and the size of its resolution proof. We have selected problems to cover different configurations:

- Large statements whose certificates do not exceed the size of the initial statement.
- Small statements for which the resolution proofs generated by state of the art solvers (Minisat, Zchaff) are quite large.

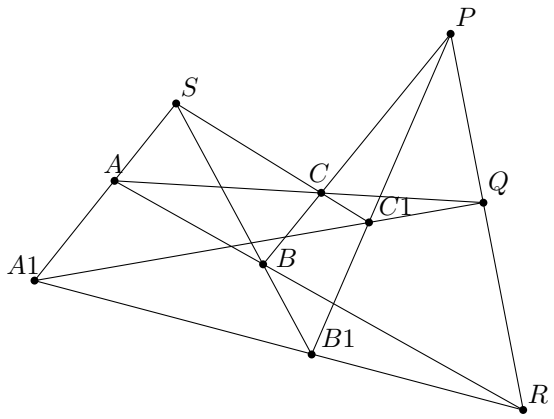
We have also selected a set of industrial examples of multipliers. These problems are known to be hard to prove.

2.7.2 Non-linear arithmetic

These benchmarks are geometrical statements that have been translated into algebra: points are represented by their coordinates, geometric predicates by polynomials based on determinants, scalar products and algebraic relations between trigonometric functions. For example, we define collinearity by:

```
Definition collinear (A B C:point):=
  (X A - X B) * (Y C - Y B) - (Y A - Y B) * (X C - X B) = 0.
```

For example, Desargues' theorem



is translated into

```
Lemma Desargues:  forall A B C A1 B1 C1 P Q R S:point,
  X S = 0 -> Y S = 0 -> Y A = 0 ->
  collinear A S A1 -> collinear B S B1 -> collinear C S C1 ->
  collinear B1 C1 P -> collinear B C P ->
  collinear A1 C1 Q -> collinear A C Q ->
  collinear A1 B1 R -> collinear A B R -> collinear P Q R
  ∨ X A = X B ∨ X A = X C ∨ X B = X C ∨ X A = 0
  ∨ collinear S B C
  ∨ parallel A C A1 C1 ∨ parallel A B A1 B1.
Proof.
geo_begin.
tzRpv 0%Z (X A::X B::Y B::X C::Y C::X A1::Y B1::Y C1::nil)
 (X B1::X C1::Y P::X P::Y Q::X Q::Y R::X R
 ::Y C1::Y B1::X A1::Y A1::Y C::X C::Y B::X B::nil).
Qed.
```

Chapter 3

Efficiency

On the particular SAT instances, the time to check in COQ the certificates based on resolution generated by external SAT solvers like MINISAT or ZCHAFF should be a tenth of time needed to generate the certificate. Resolution-based certificates are known to be sometimes huge. In order to keep the certificate small (linear with respect to the size of the initial problem), we will explore alternative certificates that are not necessarily based on resolutions. For those certificates, the criteria of success will be measured in term of size and not in term of time to produce and check certificates.

For problems that are linear, the criteria of success for the COQ part would be that the efficiency of the SMT-based approach is comparable with existing ad-hoc tactics (omega, micromega) for non-linear problem.

Few SMT solvers deal with non linearity. With respect to non-linear arithmetic, our criterion of success is to come up with pragmatic and specialised procedures able to discharge verification conditions such as those described in Section 2.6.2 and the geometric problems described in Section 2.7.2.

For proof obligations generated from the result of static analyses, SMT provers should be able to discharge and generate proofs for them using less time than it takes to check a fixpoint of the static analysis. This is more challenging than it seems because abstract domains are very specialised and can exploit additional information obtained during the fixpoint computation in order to obtain speed up. For instance, preliminary studies show that a naive SAT encoding of BDD based points-to analyses generate huge SAT proof-obligations that SAT solvers are unable to discharge.

Regarding Frama-C and Caveat proof obligations, the default amount of time given to each prover for a proof obligation is 10 seconds. Larger values, up to 1 minute, might be admissible for a few formulas, but a attaining a 10 second limit for a large majority of the proof obligations should be the objective. The following three tests are representative of the formulas generated during verification of a C programs, and it would be interesting to see if they can be solved quickly (*i.e.* less than 10 seconds) at the end of DECERT:

- `FramaC/caveat/caveat_tmp_file.06X7f9.why`
- `FramaC/cert/strdup_m_ensures_ok_po199.why`
- `FramaC/frama-c-regtests/union/union_po113.why`

The first one is a typical example of a goal generated by caveat from embedded code. The second one is generated by the jessie plugin of Frama-C during the verification of absence of run-time error in the CERT managed string library [3]. The last one is extracted from Frama-C's test base and deals with lower level features of C (namely the use of unions).

With respect to the veriT proof obligations from verification of lock-free algorithms, we expect to have validation times of a fraction of a second. The challenge is to find the right instantiation heuristics, and the resulting formula would be some typical but small SMT formulas. The TLA+ verification conditions should be small, but they are written in a very expressive language and this may require some very complex reasoning. An acceptable time would be no more than a minute.

As concerns the benchmarks coming from the Rodin platform, it is expected that these lemmas are discharged in a fraction of a second as they are very small. In a larger setting, where a lemma contains a lot of irrelevant hypotheses, this might have an influence on the time needed to discharge. However, our goal is to prevent these irrelevant hypotheses to introduce too much noise and to limit the increase of processing time.

Chapter 4

Proof Witnesses

The various parties of the consortium show different interests in proof production, and these different interests lead to different requirements concerning the form of the proofs.

With respect to Frama-C there is *a priori* few issues regarding proof witnesses. Proof witnesses should not become exceedingly large, but need not remain small at all costs. The main concern would rather be to favor the acceptance of a machine-checked proof by a certification authority. This is far beyond the scope of the DECERT project in itself, but some considerations that would ease the process might be taken into account. In this context, the main interest of having proof witnesses is the ability to separate proof search and proof verification, and to argue that only the latter must be validated according to the certification process. It would thus be important to have proof-checkers completely independent from automated theorem provers.

The Rodin platform and its plugin facility trusts the external provers; a detailed proof of the verification conditions is not required, and is not useful. However, since users develop and modify the Event-B models in parallel with attempts to prove the verification conditions that are generated from these models, very similar proof obligations will have to be reproved several times. An efficient heuristic used inside Rodin is to memorize for each valid verification condition the hypotheses that are used in its proof. External provers cooperating with Rodin should thus compute this information for every verification condition they can prove. This information can be seen as a very lightweight proof trace. The time, space, and algorithms required to reproduce the full proof from this lightweight proof trace are not a concern since this work will never have to be done.

At another extreme one find skeptical proof assistants like Isabelle (that requires proof reconstruction before admitting a lemma or theorem) and Coq (that have explicit proof objects). They may delegate verification conditions to external prover, but expect a full, detailed proof of these formulas. This proof will be replayed and every step will be checked before accepting the formula as valid. However, the size of the proof witness can be a limiting factor:

- Proof objects need to be typechecked, so before processing the proof witness, some time is dedicated to typecheck it.
- Libraries in proof assistant contains hundreds of proofs. Each time a library is used, the complete set of proof objects needs to be loaded in memory. Thus, to be able to build large hierarchy of libraries, proof objects should be kept small.

The trade-off in order to keep proofs small is to spend more time in the generation of certificates and their verification.

In a Proof Carrying Code context, proofs are a central object. They must be as small as possible and checked as fast as possible. Unlike the previous applications of SMT proofs, in a PCC setting, it is valuable to spend a lot of time optimising proofs for both size and speed. Since checking the proofs may be done on embedded processors with limited memory and processing power, to get a working PCC architecture, we consider that the size of the PCC proof should not exceed the size of the program and that the checking time has to be linear in the size of the program. The objectives concerning PCC are very ambitious. Actually, theoretical results about proof sizes are pretty negative. Yet those results are about worst cases, and we are confident those will not show up in practice.

As some preliminary works inside the veriT solver, the solver produces low level proofs for some parts of its language, namely boolean logic, possibly with uninterpreted symbols. For instance, for the toy formula in the SMT-lib language:

```
(benchmark qfuf_3
  :status unsat
  :logic QF_UF
  :extrafuns ((a U) (b U) (c U) (f U U))
  :extrapreds ((p U))
  :formula
  (and
    (= a c)
    (= c b)
    (or (not (= (f a) (f b)))
        (and (p a) (not (p b)))))
)
```

the prototype proof-producing implementation outputs the following proof:

```
1:(input ((and (= a c) (= b c) (or (not (= (f a) (f b))) (and (p a) (not (p b)))))))
2:(and ((= a c)) 1 0)
3:(and ((= b c)) 1 1)
4:(and ((or (not (= (f a) (f b))) (and (p a) (not (p b))))) 1 2)
5:(and_pos ((not (and (p a) (not (p b)))) (p a)) 0)
6:(and_pos ((not (and (p a) (not (p b)))) (not (p b))) 1)
7:(and_neg ((and (p a) (not (p b))) (not (p a)) (not (not (p b)))))
8:(or ((not (= (f a) (f b))) (and (p a) (not (p b)))) 4)
9:(eq_congruent ((not (= b a)) (= (f a) (f b))))
10:(eq_transitive ((not (= b c)) (not (= a c)) (= b a)))
11:(resolution ((= (f a) (f b)) (not (= b c)) (not (= a c))) 9 10)
12:(resolution ((= (f a) (f b))) 11 2 3)
13:(resolution ((and (p a) (not (p b)))) 8 12)
14:(resolution ((p a)) 5 13)
15:(resolution ((not (p b))) 6 13)
16:(eq_congruent_pred ((not (= b a)) (p b) (not (p a))))
17:(eq_transitive ((not (= b c)) (not (= a c)) (= b a)))
18:(resolution ((p b) (not (p a)) (not (= b c)) (not (= a c))) 16 17)
19:(resolution () 18 2 3 14 15)
```

Without going into details of this ad-hoc and temporary proof format, lines 1 to 8 account for the conjunctive normal form. Lines 8-10 and 16-17 represent the theory reasoning on uninterpreted symbols, and the remaining (resolution) rules justify the boolean reasoning as well as the articulation between the theory reasoner and the boolean reasoner.

The benchmarks provided in this first task will also serve

- as goals for developing the capabilities of the provers for producing proofs;
- to find the right level of detail the proofs should contain, according to the application;
- to investigate proof compression techniques.

Bibliography

[1] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie J. Hendren, and Navindra Umanee. Points-to analysis using bdds. In *PLDI*, 2003.

[2] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. Abcd: eliminating array bounds checks on demand. In *PLDI 2000*, pages 321–333. ACM, 2000.

[3] CERT. Managed string library. `http://www.cert.org/secure-coding/managedstring.html`.

[4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixpoints. In *Popl'77*, pages 238–252. ACM Press, 1977.

[5] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of the 5th ACM Symp. on Principles of programming languages*, pages 84–96. ACM Press, 1978.

[6] J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.

[7] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576580 and 583, October 1969.

[8] Jean-François Couchot and Thierry Hubert. A Graph-based Strategy for the Selection of Hypotheses. In *"FTP 2007 - International Workshop on First-Order Theorem Proving"*, 2007.

[9] Leslie Lamport. *Specifying Systems*. Addison-Wesley, Boston, Mass., 2002.

[10] O. Lhoták and L. J. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1), 2008.

[11] Mikel Luján, John R. Gurd, T. L. Freeman, and José Miguel. Elimination of java array bounds checks in the presence of indirection. In *JGI '02*, pages 76–85. ACM, 2002.

[12] Jia Meng and Lawrence C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *Journal of Applied Logic*, 7(1):41–57, 2009.

[13] A. Miné. The octagon abstract domain. In *Proc. of Working Conf. on Reverse Engineering 2001*, IEEE, pages 310–319. IEEE Computer Society, October 2001.

[14] Yannick Moy. *Automatic Modular Static Safety Checking for C Programs*. PhD thesis, Universit Paris Sud, 2009.

[15] Feng Qian, Laurie J. Hendren, and Clark Verbrugge. A comprehensive approach to array bounds check elimination for java. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 325–342, London, UK, 2002. Springer-Verlag.

[16] F. Randimbivololona, J. Souyris, P. Baudin, A. Pacalet, J. Raguideau, and D. Schoen. Applying formal proof techniques to avionics software: A pragmatic approach. In *FM'99*, 1999.

[17] Silvio Ranise and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `http://www.smtcomp.org`, 2006.

[18] Richard Bornat. Proving Pointer Programs in Hoare Logic. In *Mathematics of Program Constructions*, 2000.

[19] Frama-C Development team. Frama-c. `http://frama-c.cea.fr`.

[20] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proc. of the ACM conference on Programming language design and implementation (PLDI '04)*, pages 131–144. ACM, 2004.

[21] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Array bounds check elimination for the java hotspot$^{\text{TM}}$client compiler. In *PPPJ '07*, pages 125–133. ACM, 2007.