Automated Formal Methods (AFM09)

Fourth Workshop, Grenoble, France, June, 2009

Hassen Saïdi, Natarajan Shankar (Eds.)

Preface

The Fourth Workshop on Automated Formal Methods (AFM) was held on June 27, 2009 as part of the 21st International Conference on Computer Aided Verification (CAV) 2009 in Grenoble, France. The workshop is a forum for the discussion of the development, use, and integration of automated tools for formal methods. The topics of discussion are centered around, but not exclusive to, the SRI suite of tools (PVS, SAL and HybridSAL, and Yices), and experiences and experiments involving the application, integration, and extension of these and similar tools.

The AFM workshop lays special emphasis on the use of highly automated tools for formal methods and the tight or loose integration of different methods into tool chains that strive to provide end-to-end automation for all aspects of software development. Users of all levels of expertise are invited to report on their experiences.

The workshop program included six contributed papers and four invited tutorials selected by the international program committee.

The invited tutorials included

Hassen Saïdi (SRI International): *Challenges in analyzing binary programs*
Abstract: Program analysis is an enough challenging task when source code is available. It is even more challenging when neither the source code nor debug information is present. The analysis task is further hindered when the available binary code has been obfuscated to prevent the analysis from being carried out. In this presentation, we review the main challenges when analyzing binary programs and explore techniques for recovery of information that allows program understanding and reverse-engineering. We illustrate these techniques on the Conficker worm that has plagued the Internet in the past few months.

Susanne Graf, Verimag: *Contracts for the component-based design of embedded and distributed systems*
Abstract: Distributed, real-time and embedded systems usually multiple layers from the high-level functional layers down to the interaction with hardware. The design of such systems leads to complex hierachical architectures with components subject to multiple constraints. The BIP composition operators allow specifying complex multi-party interactions between components in a hierarchical fashion, and by separating component behaviour and interaction between components. It is expressive enough to describe the interaction of a set of peers so as to abstract lower layers as composition operator represented by a set of connectors and their interactions. We define a notion of contract associated with components which strictly separates an expectation which it may have on the environment, called *assumption*, and a *promise* which is behaviour of the component under consideration that the environment may take for granted as long as it respects the component's expectation. Contrary to most notions of contracts, it does not express the assumptions directly on the component's interface but as a constraint on its

peers to which it is connected by a rich set of connectors as in BIP. We do not intend contracts to be used for compositional verification but rather for compositional design and independent implementation of components. Assumptions allow simplifying component implementations by relying on properties ensured by the environment. An interesting of our kind of contracts is to allow expressing also assumptions which need not to be expressible on the component's interface. This means that the component interfaces need not to be "artificially" enriched with analysis related attributes. Moreover, knowledge about peers and about lower layers is clearly separated, and specifications of lower layers, represented by a set of connectors, may be refined independently of components. So far, we have shown that this general contract framework is indeed a generalisation of all existing notions of interface specifications or contracts that we have studied and proposed some general methodology. Here, we propose also a set of useful concepts which can be used to actually express contracts for components which must comply to safey and progress constraints.

Bernhard Steffen (TU Dortmund): *Continuous Model-Driven Engineering - Formal Methods at the Application Level*

Abstract: Agility is a must, in particular for business-critical applications. Complex systems and processes must be continuously updated in order to meet the ever changing market conditions. Continuous Model Driven Engineering addresses this need by by continuously involving the customer/application expert throughout the whole systems? life cycle including maintenance and evolution. Conceptually, it is based on the One Thing Approach (OTA), which combines the simplicity of the waterfall development paradigm with a maximum of agility. The key to OTA is to view the whole development process simply as a complex hierarchical and interactive decision process, where each stakeholder, including the application expert, is allowed to continuously place his/her decisions in term of constraints. Thus semantically, at any time, the state of the development or evolution process can simply be regarded as the current set of constraints, and each development or evolution step can be regarded simply as a transformation of this very constraint set. This approach, conceptually, allows one 1) to monitor globally and at any time the consistency of the development or evolution process simply via constraint checking, and 2) to impose a kind of decision hierarchy by mapping areas of ompetencies to roles of individuals, in order to identify required actions in case of constraint violation. The essence and power of this approach, which is technically supported by the jABC development and execution framework, will be illustrated along real life application scenarios.

Jean Christophe Filliâtre, LRI: *Why - an intermediate language for deductive program verification*

Abstract: This tutorial is an introduction to the Why tool, an intermediate language for deductive program verification. The purpose of the Why tool is two-fold: first, it computes weakest preconditions for a small alias-free programming language, which is designed to be the target of other verification tools for languages such as C or Java; second, it translates verification con-

ditions into the native languages of several existing theorem provers, either automatic such as Simplify, Alt-Ergo, Yices, Z3, etc. or interactive such as Coq, PVS, Isabelle, etc. Why is currently used in several verification frameworks such as Caduceus, Krakatoa, or Frama-C.

There were six contributed papers.

- **Alwyn Goodloe, Corina Păsăreanu, David Bushnell and Paul Miner.** *A Test Generation Framework for Distributed Fault-Tolerant Algorithms*
- **Anduo Wang and Boon Thau Loo.** *Formalizing Meta-Routing in PVS*
- **Ilya Lopatkin, Daniel Plagge, Alexei Iliasov, Michael Leuschel and Alexander Romanovsky.** *SAL, Kodkod, and BDDs for Validation of B Models*
- **Luca Chiarabini.** *Automatic Synthesis of an Efficient Algorithm for the Similarity of Strings Problem*
- **Silvio Ghilardi and Silvio Ranise.** *Model-Checking Modulo Theories at Work: The integration of Yices in MCMT*
- **Jean-Francois Couchot, Alain Giorgetti and Nicolas Stouls.** *Graph-based Reduction of Program Verification Conditions*

Hassen Saïdi
Natarajan Shankar
Menlo Park, California

Programme Chairs

Hassen Saïdi
Natarajan Shankar

Programme Committee

Myla Archer
Saddek Bensalem
Aaron Bradley
Supratik Chakraborthy
Rance DeLong
Jean-Christophe Filliâtre
Bernd Finkbeiner
Michael Gordon
John Harrison
Peter Manolios
David Monniaux
Leonardo de Moura
David Naumann
Kazuhiro Ogata
Corina Păsăreanu
Lee Pike
Sanjit Seshia
Ofer Strichman

External Reviewers

Timothy G. Griffin — University of Cambridge

Table of Contents

Formalizing Metarouting in PVS 1
Anduo Wang, Boon Thau Loo

A Test Generation Framework for Distributed Fault-Tolerant Algorithms 8
Alwyn Goodloe, Corina Păsăreanu, David Bushnell, Paul Miner

SAL, Kodkod, and BDDs for Validation of B Models 16
Daniel Plagge, Ilya Lopatkin, Alexei Iliasov, Michael Leuschel, Alexander Romanovsky

Automatic Synthesis of an Efficient Algorithm for the Similarity of Strings Problem 23
Luca Chiarabini

Model-Checking Modulo Theories at Work: the integration of Yices in MCMT 31
Silvio Ghilardi, Silvio Ranise

Graph-based Reduction of Program Verification Conditions 40
Jean-Francois Couchot, Alain Giorgetti, Nicolas Stouls

Formalizing Metarouting in PVS

Anduo Wang
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, USA
anduo@seas.upenn.edu

Boon Thau Loo
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, USA
boonloo@seas.upenn.edu

ABSTRACT

In this paper, we extend PVS specification logic with abstract *metarouting* theory to aid the development of complex routing protocol models based on *metarouting*, which is an algebraic framework for specifying routing protocols in a restricted fashion such that the protocol is guaranteed to converge. Our formalization of metarouting theory utilizes the *theory-interpretation* extensions of PVS. Our use of a general purpose theorem prover provides a structured framework for a network designer to incrementally develop and refine their algebraic routing protocol model by starting from various base routing algebras, and composing them into complex algebra models with composition operators. In addition, one can leverage PVS's type checking capability and built-in proof engine to ensure routing model consistency.

1. INTRODUCTION

The Internet today runs on a complex routing protocol called the *Border Gateway Protocol* or *BGP* for short. BGP enables Internet-service providers (ISP) world-wide to exchange reachability information to destinations over the Internet, and simultaneously, each ISP acts as an autonomous system that imposes its own import and export policies on route advertisements exchanged among neighboring ISPs.

Over the past few years, there has been a growing concern on the complexity and fragility of BGP routing. Even when the basic routing protocol converge, conflicting policy decisions among different ISPs have lead to route oscillation and slow convergence. Several empirical studies such as [7] have shown that there are prolonged periods in which the Internet cannot reliably route data packets to specific destinations due to routing errors induced by BGP. In response, the networking community has proposed several Internet architectures and policy mechanisms (e.g. [1]) aimed at addressing these challenges.

Given the proliferation of proposed techniques, there is a growing interest in formal software tools and programming frameworks that can facilitate the design, implementation, and verification of routing protocols. These proposals can be broadly classified as: (1) algebraic and logic frameworks (e.g. [3]) that enable protocol correctness checking in the design phase; (2) runtime debugging platforms that provide mechanisms for runtime verification and distributed replay, and (3) programming frameworks that enable network protocols to be specified, implemented, and in the case of the Mace toolkit, verified via model checking [6].

In this paper, we extend PVS specification logic with abstract *metarouting* theory [3] to aid the development of complex routing protocol models based on *metarouting*, which is an algebraic framework for specifying routing protocols in a restricted fashion such that the protocol is guaranteed to converge. Using the *theory-interpretation* [8] extensions of the PVS theorem prover, we formalize in PVS a variety of metarouting algebra instances and demonstrate that an interactive theorem prover is suitable for modeling the complicated BGP system using the metarouting theory developed in PVS.

The main benefits of formalizing metarouting within a mechanized theorem prover are as follows. First, the network designer can now focus on high-level protocol design and the conceptual decomposition of the BGP system, and shift the low level details of ensuring consistency of the derived protocol model with respect to metarouting theory to the PVS type checker. Second, the PVS proof engine handles most of the proof effort (via the top-level strategy *grind* and other built-in type checking capabilities), and therefore frees the network operator from the trivial and tedious proof necessary to ensure the convergence of their BGP algebra model. In the long run, we believe that our framework will also result in the support of relaxed algebra models, which allow a wider range of well-behaved convergent component protocols to be supported compared to the restrictions imposed by metarouting.

2. BACKGROUND

2.1 Internet Routing

The Internet can be viewed as a network of *Autonomous Systems* (AS) each administrated by an *Internet Server Provider* (ISP). The *routing protocol* is executed on all ASes in order to compute reachability information. Given a destination address, each packet sent by a source is forwarded by each intermediate node to the next neighboring node along the best

path computed by the routing protocol.

In particular, within an AS, the ISP runs its own class of routing protocols called the *Internal Gateway Protocol* (IGP), whereas between ASes , the class of protocols used are called the *External Gateway Protocol* (EGP). EGP enables routing across AS administration borders by including mechanism for *policy-based routing*. The role of policy routing is to allow ISPs to influence route decisions for economical or political concerns, and the basic mechanism used is to decide which routes to accept from neighbors (*import policies*), and which routes to advertise to other neighbors (*export policies*).

2.2 Metarouting

The Internet uses the *Border Gateway Protocol* (BGP) as its de facto routing protocol. This protocol is a combination of the IGP/EGP protocol described above. Metarouting [3] is first proposed to extend the use of routing algebra to BGP design and specification. Metarouting enables the construction of a complicated BGP system model from a set of pre-defined base routing algebras and composition operators. Prior to metarouting, Griffin *et al.*first proposed combinatorial models for BGP [2, 4] to aid the static analysis of convergence of routing protocols. Later a *Routing Algebra Framework* was proposed by Sobrinho [9, 10] to provide the rigorous semantics for the design and specification of routing protocols. Sobrinho uses various algebra instances to represent possible routing protocols and policy guidelines. Sobrinho further identifies and proves monotonicity as a sufficient condition for protocol convergence. Meta-routing builds upon these two earlier pieces of work. In the rest of the section, we provide a short overview of metarouting.

First, metarouting adopts the use of routing algebra as the mathematical model for routing. An abstract routing algebra is a tuple A: $A = \langle \Sigma, \preceq, \mathcal{L}, \oplus, \mathcal{O}, \phi \rangle$. Here Σ is the set of *signatures* used to describe paths in the network totally ordered by preference relation $\preceq$. Intuitively, the preference relation is used by a routing protocol to optimize path cost; $\mathcal{L}$ is a set of $labels$ describing links between immediate neighbors. Note that labels may denote complicated policies associated with the corresponding link; $\oplus$ is a mapping from $\mathcal{L} \times \Sigma$ to Σ, which is the *label application operation* that generates new paths by combining existing paths and adjacent links; And $\mathcal{O}$ is a subset of Σ called *origination* that represents the initial routes stored at network nodes; Finally ϕ is a special element in Σ denoting the prohibited path. The semantics of routing algebra is given by the following axioms:

Maximality	$\forall_{\alpha \in \Sigma - \{\phi\}} \quad \alpha \preceq \phi$
Absorption	$\forall_{l \in \mathcal{L}} \quad l \oplus \phi = \phi$
Monotonicity	$\forall_{l \in \mathcal{L}} \forall_{\alpha \in \Sigma} \quad \alpha \preceq l \oplus \alpha$
Isotonicity	$\forall_{l \in \mathcal{L}} \forall_{\alpha, \beta \in \Sigma} \quad \alpha \preceq \beta \implies l \oplus \alpha \preceq l \oplus \beta$

Maximality and **Absorption** are straightforward properties of the prohibited path ϕ, stating that any other paths are always preferred over ϕ, and that extending the un-usable path ϕ with any usable link would still result in prohibited path. On the other hand, **Monotonicity** and **Isotonicity** are two non-trivial properties that ensure network *convergence*[1] of a routing protocol modeled by the routing algebra.

Furthermore, based on the abstract routing algebra, metarouting identifies a set of atomic (base) algebras such as *ADD(n,m)* and *LP(n)*, and composition operators such as *Lexical Product* $\otimes$ and *Scaled Product* $\odot$ as the building blocks for more complicated routing algebras. This paper presents the incremental development of metarouting abstract algebras and the use of such abstract theory to build concrete BGP systems.

Unlike previous combinatorial models [2, 4], metarouting identifies and proves that the properties of *monotonicity* and *isotonicity* are sufficient conditions for network convergence. Convergence verification of BGP systems are then reduced to proofs of monotonicity and isotonicity of the related routing algebra, whereas in the analysis of BGP systems using previous combinatorial models, the proof requires genuine insights into the models themselves.

Despite its advantages, metarouting is fairly restricted in two ways. First, it cannot represent all protocols that converge. Second, it places the burden on network designers to write algebras and composition operators correctly. Our work aims to address these two limitations by using PVS to provide a framework for expressing routing algebras and their operators correctly, and then flexibly reason about the convergence properties of these protocols even when the sufficient conditions are violated. One should view our paper as providing the initial building blocks and methodology for interesting explorations elaborated in Section 5.

3. BASIC APPROACH

This section describes the basic technique of embedding metarouting in PVS. In particular, this paper presents the development of metarouting in PVS using its extensions on *theory interpretation* [8].

The basic approach is to encode metarouting algebraic objects in PVS's type system. It involves formalization of abstract routing algebra theory A, the set of atomic algebra instances of A, and composition operator $\otimes$.

First of all, the abstract routing algebra structure $A = \langle \Sigma, \preceq, \mathcal{L}, \oplus, \mathcal{O}, \phi \rangle$ is formalized as an uninterpreted abstract (source) theory in PVS, as described in [11, 5, 8]. The basic idea is to use *types* to denote the sets of objects $\Sigma, \mathcal{L}, \mathcal{O}$. Accordingly, the special element ϕ denoting prohibited path is expressed as an uninterpreted constant of type Σ. And the preference relation $\preceq$ and the label application operation are denoted by functions. The PVS theory for abstract route algebra A is given as follows:

[1] A network routing protocol converges when all routing tables can be computed to a distributed fixpoint given a stable network, and when any links are updated, these routing tables can be incrementally recomputed similarly to a fixpoint.

```
routeAlgebra: THEORY
  BEGIN
  sig: TYPE+
  prefRel: [sig, sig -> bool]
  label: TYPE+
  labelApply: [label, sig -> sig]
  prohibitPath: sig
  initialS: [sig -> bool]
  org: TYPE = s: sig | initialS (s)
  END routeAlgebra
```

Here uninterpreted types `sig` and `label` denote sets Σ and $\mathcal{L}$, and `org` denoting initial route set $\mathcal{O}$ is made subtype of Σ via auxiliary predicate `initialS` which decides if a path falls into the initial routes. Finally ϕ is made constant `prohibitPath` of type `sig`.

Semantics of abstract routing algebra A are given by the following axiomatic specification:

```
monotonicity: AXIOM
  FORALL (l: label, s: sig): mono(l, s)

isotonicity: AXIOM
  FORALL (l: label, s1, sig, s2: sig):
    prefRel(s1,s2) =>
      prefRel(labelApply (l,s1),labelApply(l,s2))

maximality: AXIOM
  FORALL (s: sig): prefRel (s, prohibitPath)

absorption: AXIOM
  FORALL (l: label):
    labelApply (l, prohibitPath) = prohibitPath
```

Where `eqRel` (`eqRel` will be used later) and `mono` are defined by the following auxiliary predicates:

```
eqRel (s1, s2: sig): bool =
  prefRel (s1, s2) and prefRel (s2, s1)
mono(l: label, s: sig): bool =
  prefRel (s, labelApply (l, s))
```

Note that this abstract routing theory A then stands for all possible routing algebra instances. We also observed that PVS parametric theories offer an alternative to define abstract algebra, sketched as follows:

```
routeAlgebra[sig: TYPE+,
             prefRel: [sig, sig -> bool],
             label:TYPE+,
             labelApply: [label, sig -> sig]]
 BEGIN
  prohibitPath: sig
  initialL: [label -> bool]
  org: TYPE+ = l: label | initialL (l)
  ...
 END routeAlgebra
```

In the rest of this paper, to exploit PVS's theory interpretation mechanism, we will use of the first uninterpreted theory representation. To encode the basic building blocks of metarouting: atomic routing algebras and composition operators, we utilize two features provided by the PVS theory interpretation [8] extensions: *mapping* and *declaration*. With the *mapping* mechanism, a general (source) theory is instantiated to an interpretation (target) theory. On the other hand, the *theory declaration* mechanism takes PVS theories as parameters, and therefore, unlike mapping, can build a theory from multiple source structures.

Figure 1 shows our basic two-step approach to formalize metarouting building blocks. First, we utilize abstract routing algebra theory `A` developed above as a source theory, applying PVS's *mapping* mechanism to this source theory to yield the set of interpretation theories `Ii` for atomic routing algebras I_i. Second, by applying PVS's theory *declaration* mechanism, we encode composition operator O_i as PVS theories taking routing algebra as parameters, which can be further instantiated to yield the resulting compositional routing algebra `Oi`.

The main benefit of this approach is that the semantics (axioms) of source routing theory `A` are enforced automatically in all target theories `Ii` and `Oi`. This ensures that all atomic routing algebra instances are valid routing algebras and that all composition operators are closed under abstract routing algebra (i.e. any compositional routing algebras that can be derived using operators `Oi` are guaranteed to be routing algebras as defined by the abstract routing algebra theory). The detailed formalization of metarouting building blocks using PVS theory interpretation is presented in the next section.

4. COMPOSITIONAL ROUTING ALGEBRA

This section presents the formalization of metarouting building blocks by stepping through atomic routing algebras *addA(n,m)* and *cpA(n)*; as well as composition operator lexical product $\otimes$.

4.1 Atomic Routing Algebra Instance

Shortest Path Routing.

The first simple routing algebra *addA(n,m)* describes shortest path routing. The labels/signatures can be thought of as the distance costs associated with the corresponding links/paths. Note that in practice, costs of valid links/paths have an upper bound, and links/paths with higher cost are considered prohibited. We use PVS theory `addA` to capture algebra *addA(n,m)* as follows:

```
addA: THEORY
 BEGIN
  n, m: posnat
  N_M: AXIOM n < m
  LABEL: TYPE = upto(n)
  SIG: TYPE = upto(m + 1)
  ...
 END addA
```

Here `n, m` are the uninterpreted constants denoting link/path cost bounds. The preference relation $\preceq$ over signatures `SIG` are then simply interpreted as the normal $\leq$ relation over natural numbers, indicating that a low-cost path is preferred over high-cost path. The label application operation $\oplus$ can be interpreted as a function that computes the cost of the new path obtained from a sub-path and the adjacent link, where

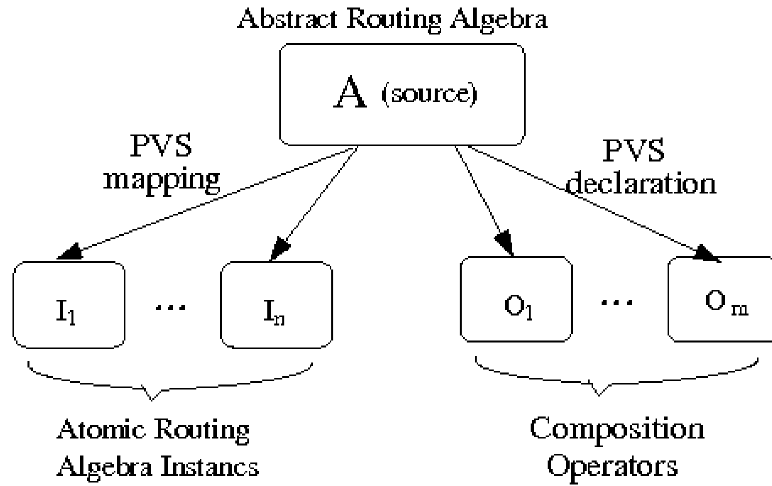


Figure 1: Overview of PVS Theories

the cost of the new path is simply the normal addition of that of the sub-path and link. In PVS, we write as follows:

```
PREF(s1, s2: SIG): bool = (s1 <= s2)
APPLY(l: LABEL, s: SIG): SIG =
  IF (l+s < m+1) THEN (l+s) ELSE (m+1) ENDIF
```

Note that in the definition of label application function `APPLY`, `m+1` is used as the value of prohibited path. This is directly defined using PVS mapping of abstract algebra `routeAlgebra` in the following `IMPORTING` clause:

```
IMPORTING
  routeAlgebra{{sig := SIG,
                label := LABEL,
                prohibitPath := m + 1,
                labelApply(l:LABEL, s:SIG)
                             := APPLY(l,s),
                prefRel(s1, s2: SIG)
                             := PREF (s1, s2)}}
```

Recall that a routing algebra consists of a set of signatures `sig`, labels `label`, preference relations `prefRel` over signatures, and label application functions `labelApply`. Here, theory `addA` imports the uninterpreted abstract algebra theory `routeAlgebra`, and makes the following instantiations:

$$\begin{aligned} \texttt{sig} &\leftarrow \texttt{upto}(\texttt{m}+1) \\ \texttt{label} &\leftarrow \texttt{upto}(\texttt{n}) \\ \texttt{prohibitPath} &\leftarrow \texttt{m}+1 \\ \texttt{labelApply} &\leftarrow \texttt{APPLY} \\ \texttt{prefRel} &\leftarrow \texttt{PREF} \end{aligned}$$

The corresponding instances of `routeAlgebra` axioms defining the semantics of routing algebra are proof obligations called type correctness conditions (*TCCs*). For example, monotonicity axiom is instantiated and denoted by the following automatically generated TCC:

```
IMP_A_monotonicity_TCC1: OBLIGATION
  FORALL (l: LABEL, s: SIG): mono(l, s)
```

All of the TCCs are automatically discharged by either the default TCC proof strategy or high-level strategy `grind`.

So far, we have established the encoding of shortest path algebra in PVS by providing mappings for uninterpreted types in the source theory `routeAlgebra` into the target theory (interpreting) `addA`. We observe that even in this simple example, PVS significantly reduces manual effort ensuring consistency, generating proof obligations and enabling the user to focus on high-level mapping for shortest path routing.

Customer-Provider and Peer-Peer Relationship. We provide another example of base/atomic algebra *cpA(n)* that captures the policy guideline regarding the economic relationship between ASes. Customer-Provider and Peer-Peer relationships between ASes are prevalent in today's Internet. A common policy guideline to help BGP convergence is to always prefer customer-routes to peers or providers routes.

More specifically, in the algebra $cpA = \langle \Sigma, \preceq, \mathcal{L}, \oplus, \mathcal{O}, \phi \rangle$, the signature set can take three values $C/R/P$, representing customer/peer/provider routes respectively (i.e. routes advertised by a node's customer, peer, or provider). Accordingly, labels can take values $c/r/p$, representing customer/peer/provider link (i.e. links to customer/peer/provider).

The preference relation over signatures is given by: $C \preceq R, \quad R \preceq P, \quad C \preceq P$. Intuitively this relation means, a customer route is always preferred over a peer and provider route, and a peer route is preferred over a provider route. The intuition is that each ISP enforces the policy to reduce the use of provider routes, while maximizing availability and use of its customer routes.

The complete definition of the label application operation $\oplus$ is given by the following table:

$\oplus$	C	R	P
c	C	C	C
r	R	R	R
p	P	P	P

For example the first line $c \oplus (C/R/P) = C$ can be read as a customer/peer/provider path extended by a customer link results in a customer path, hence has the highest priority of all available paths.

For simplicity, rename labels and signatures as follows: $c \leftarrow 1, r \leftarrow 2, p \leftarrow 3$ and $C \leftarrow 1, R \leftarrow 2, P \leftarrow 3$. This renaming enables the preference relation to be expressed as normal $\leq$ over natural number. Similar to the algebra for shortest path, `cpA` can be encoded using PVS mapping as follows:

```
cpA: THEORY
  BEGIN
  SIG: TYPE = x: posnat | x<=3
  LABEL: TYPE = x: posnat | x<=3
  APPLY (l: LABEL, s: SIG): SIG = l
  IMPORTING
    routeAlgebra{{sig := SIG,
                  label := LABEL,
                  labelApply(l:LABEL, s:SIG)
                             := APPLY (l,s),
                  prohibitPath := c+1}}
  END cpA
```

As in the case of shortest paths, all the TCCs enforcing routing algebra axioms (for example, monotonicity) are automatically discharged. This is consistent with the intuition that customer-provider policy does help BGP convergence.

4.2 Lexical Product and Route Selection

This section presents development of lexical product $\otimes$, a composition operator that enables construction of routing algebra from atomic algebra described in section 4.1. It is particularly useful in modeling route selection in BGP system where multiple attributes are involved.

Consider product algebras $A \otimes B$ constructed from two route algebra `A, B`, where the parameter theories `A` and `B` model two attributes `a` and `b` respectively. First define the signature and label of $A \otimes B$ as product of that from `A` and `B` in PVS as:

```
lexProduct[A, B: THEORY routeAlgebra]: THEORY
  BEGIN
  SIG: TYPE = [A.sig, B.sig]
  LABEL: TYPE = [A.label, B.label]
  ...
  END lexProduct
```

Here the first component of signature/label comes from `A` and the second component comes from `B`. And a natural interpretation of label application function over path and label is given by the following product in PVS:

```
APPLY(l:LABEL,s:SIG):SIG =
(A.labelApply(l`1,s`1),B.labelApply(l`2,s`2))
```

Here the two components invoke the corresponding label application functions defined in theory `A` and `B` respectively.

Next consider the preference relation over $A \otimes B$ that in PVS as follows:

```
PREF(s1,s2:SIG):bool =
 A.prefRel(s1`1, s2`1) OR
 (A.eqRel(s1`1,s2`1) AND B.prefRel(s1`2,s2`2))
```

The above definition is particularly interesting because it models the route selection process in BGP system. This preference relation reads as: a path with two attributes `a` and `b` represented by signature `s1` is considered better than a path denoted by `s2` given one of the two following conditions: (1) first component `s1`1` of `s1` is better than the first component `s2`1` of `s2`, as defined in algebra `A`; or (2) if the first component of `s1` and `s2` are equally good, but `s1` is better than `s2` with respect to the second component, as described in algebra `B`. This lexicographic comparison captures the route selection process, which is a major part for any BGP system with multiple attributes. Intuitively, in selecting a route towards a given destination, the router compares all its possible paths towards that destination by going through a comparison list, checking one attribute at a time, selecting the best path based on attributes ordering. The router goes down the list and compares the next attribute only if the attributes seen in previous steps are equally good.

As before, we can now instantiate route algebra theories and corresponding sets of axioms as follows:

```
IMPORTING
  routeAlgebra{{sig := SIG,
                label := LABEL,
                labelApply(l:LABEL,s:SIG)
                           := APPLY(l,s),
                prefRel(s1, s2: SIG)
                       := PREF(s1, s2)}}
```

Again, PVS automatically generate and prove all the type checking conditions.

4.3 A Concrete First Example

This section presents an concrete example routing protocol algebra built from metarouting atomic algebras and composition operators developed in previous sections. We demonstrate the ease of applying abstract metarouting theory to concrete example algebra in PVS. In particular, we highlight the intuitive networking interpretation in practice.

Consider a simple BGP system where the route paths are measured in terms of customer-provider relationship and distance cost. For all possible routes reaching a given destination, a route path going through customers and peers is preferred to path going through providers; and a route go through peers is preferred to those through providers. Once this customer-provider policy is enforced, the ISP is concerned with distance cost with respect to each path. For the same types of paths, the ISP will choose the shortest path with lowest cost.

In the top level, this BGP system can be decomposed into two sub-components: customer-provider component and the shortest path component developed in section 4.1. Because the customer-provider relationship has higher-priority over the distance cost attribute, it can be naturally implemented by construction using lexical product, as shown in the following PVS code:

```
firstExample: THEORY
 BEGIN
 IMPORTING AlgebraInstance, lexProduct
 firstAlgebra: THEORY = lexProduct[A2,B2]
 END firstExample
```

Here `firstAlgebra` is defined to be the concrete algebra modeling this BGP system. It is constructed from customer-provider component algebra `A2` and shortest path algebra `B2` by applying lexical product, where `A2` and `B2` are defined in the imported theory `AlgebraInstance`. First, we show the definition of `A2` that enforces ISP customer-provider policy simply as an instance of `cpA`, where the uninterpreted constant `c` is mapped to 3.

```
AlgebraInstance: THEORY
 BEGIN
  IMPORTING cpA{{ c := 3 }}
  A2:THEORY =
    routeAlgebra{{
      sig = cpA.SIG,
      label = cpA.LABEL,
      labelApply(l:cpA.LABEL,s:cpA.SIG)
                          = mod(l+s,c),
      prohibitPath = c + 1,
      prefRel(s1,s2:cpA.SIG) = (s1<=s2)}}
  ...
 END AlgebraInstance
```

Likewise, concrete algebra `B2` for shortest path can be defined in terms of `addA` as follows:

```
IMPORTING addA{{n:= 16, m:=16}}
B2:THEORY =
  routeAlgebra{{
    sig = addA.SIG,
    label = addA.LABEL,
    labelApply(l:int,s:int) = l+s,
    prohibitPath=16,prefRel(s1,s2:int)
                        =(s1<=s2)}}
```

Where uninterpreted bounds on signature/labels `m/n` in `addA` are mapped to 16, which is the actual value used in distance vector protocol practice. Finally, by type checking, PVS automatically figures out all type correctness conditions to ensure consistency. All of the TCCs are discharged with default/high-level proof procedure in one step. This ensures the BGP system we derived from atomic algebras `addA`, `cpA` by using composition operator $\otimes$ are indeed a valid routing algebra that is guaranteed to converge.

In summary, we observe that by incorporating metarouting abstract theory, a non-specialized standard proof assistant like PVS, can be used to specify a specific routing protocol instance with great ease. And the routing algebra semantics is enforced by proof obligations (TCCs) automatically generated in PVS, all of which can be discharged by either PVS default TCC proof strategy or high-level strategy `grind` in one step!

5. FUTURE WORK

A straightforward application of the specification technique we explored in this paper is to construct incrementally in PVS the routing algebraic model for complicated BGP

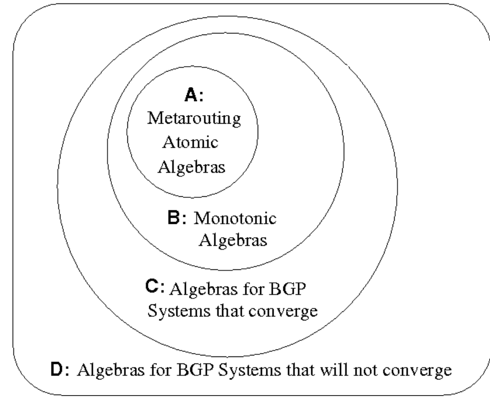


Figure 2: Classification of BGP algebras

systems by using the base algebra blocks and composition operators we developed in this paper, and the resulting algebraic model checked in PVS is then used as part of design document to derive the real BGP implementation. To achieve full set support for the modeling of BGP system via metarouting, we plan to encode in PVS more base routing algebras, such as `TAG` which is critical in the modeling of complicated routing policies, and more composition operators, such as scoped product, which models a BGP system running in and between administrative regions (i.e. the behavior of BGP protocol across AS boarder).

Furthermore, we conceive a more ambitious (adventurous) use of PVS to aid the verification of BGP system convergence using a relaxed algebra model. As depicted in Figure 2, we label the set of atomic metarouting algebras with type `A` and denote them with the inner ring. We then observed that all metarouting algebras that can be composed from type `A` algebras by composition satisfy monotonicity by definition and therefore fall into type `B` algebras represented by the middle ring. Sobrinho's original paper [9] showed that monotonicity is a *sufficient* (not necessary) condition for BGP system convergence. There are known BGP systems that converge but violate monotonicity, and this reveals existence of type `C` algebras modeling the set of converging BGP systems that are not monotonic. By relaxing the monotonicity property, we would like to explore the modeling and reasoning of type `C` systems that fall outside Monotonic type `B` Algebra (the middle ring) but are equally good with respect to convergence. Taking this basic approach one step further, instead of starting from the algebra model, we would like to develop in PVS an algebraic representation of a given BGP system that falls outside the scope of current metarouting algebra, and with the aid of PVS proof engine, decide if that corresponding BGP system falls into type `C` and converges or type `D` that does not converge.

6. REFERENCES

[1] Ee, C. T., Chun, B.-G., Ramachandran, V., Lakshminarayanan, K., and Shenker, S. Resolving Inter-Domain Policy Disputes. In *SIGCOMM* (2007).

[2] Griffin, T. G., Shepherd, F. B., and Wilfong, G. The stable paths problem and interdomain routing. *IEEE/ACM Trans. Netw.* (2002).

[3] Griffin, T. G., and Sobrinho, J. L. Metarouting. In *ACM SIGCOMM* (2005).

[4] Griffin, T. G., and Wilfong, G. An Analysis of BGP Convergence Properties. *SIGCOMM Comput. Commun. Rev.* (1999), 277–288.

[5] Gunter, E. L. Doing algebra in simple type theory.

[6] Killian, C., Anderson, J., Jhala, R., and Vahdat, A. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI* (2007).

[7] Labovitz, C., Malan, G., and Jahanian, F. Internet Routing Instability. *ACM/IEEE Trans. on Networking* (1998).

[8] Owre, S., and Shankar, N. Theory interpretations in pvs. Tech. rep., 2001.

[9] Sobrinho, J. Network routing with path vector protocols: theory and applications. In *SIGCOMM* (2003).

[10] Sobrinho, J. An algebraic theory of dynamic network routing. Tech. rep., October 2005. (William R. Bennett Prize 2006).

[11] Windley, P. J. Abstract theories in hol. In *HOL'92: Proceedings of the IFIP TC10/WG10.2 Workshop on Higher Order Logic Theorem Proving and its Applications* (1993), North-Holland/Elsevier, pp. 197–210.

A Test Generation Framework for Distributed Fault-Tolerant Algorithms

Alwyn Goodloe
National Institute of Aerospace

Corina S. Păsăreanu
Carnegie Mellon University/NASA Ames

David Bushnell
TracLabs/NASA Ames

Paul Miner
NASA Langley

ABSTRACT

Heavyweight formal methods such as theorem proving have been successfully applied to the analysis of safety critical fault-tolerant systems. Typically, the models and proofs performed during such analysis do not inform the testing process of actual implementations. We propose a framework for generating test vectors from specifications written in the Prototype Verification System (PVS). The methodology uses a translator to produce a Java prototype from a PVS specification. Symbolic (Java) PathFinder is then employed to generate a collection of test cases. A small example is employed to illustrate how the framework can be used in practice.

1. INTRODUCTION

Verification and validation of distributed fault-tolerant systems is a continuing challenge for safety-critical systems. In order to provide V&V support for distributed fault-tolerant algorithms, we are exploring a combination of technologies. Ultimately, fault tolerance consists of establishing and maintaining consensus between distributed computational resources, especially when a bounded subset of these resources is faulty. A full analysis requires an understanding of both the distribution and failure modes of the sensors, effectors, and computational resources. There are several different valid ways of architecting these systems to meet fault-tolerance requirements. This compounds the problem of providing a collection of tools supporting V&V activities. Substantiating fault-tolerance claims requires a combination of analysis and test. We are researching an approach to V&V where the test-vectors are generated from formal models expressed using SRI's Prototype Verification System (PVS).

Given that safety-critical systems are usually developed to exacting certification criteria, system failures are often the result of unanticipated events such as dirty voltage on a bus or a hardware fault. A formal model of a fault-tolerant system should explicitly model the faults that the system can handle and a testing regime should validate that the system does indeed process these as advertised. Thus testing the actual system must include injecting faults into the system. One preferred way to do this is to employ a test harness that can inject data into the system so that it appears as if a fault has occurred. Given a PVS specification, we are developing a methodology for generating these tests automatically.

Rather than develop a new tool suite from scratch, we apply two existing tools to the task of test case generation. A PVS to Java translator, developed to create executable prototypes from the specification, is applied to generate a realization of the protocol preserving the correctness properties shown to hold in the PVS model. Symbolic Java PathFinder, a model-based automated software test generation tool [9], is then used to generate test vectors that can be used by V&V engineers to test actual protocol implementations.

The paper is organized as follows. We first introduce a case study of a failure in the space shuttle. The next section provides an overview of fault-tolerance. This is followed by an overview of a PVS model of a small consensus protocol. Next is a brief description of the PVS-to-Java translator. We then discuss Symbolic Java PathFinder. Section 7 discusses test case generation for the protocol. Finally, we discuss related works and conclude.

2. FAILURE IN THE SPACE SHUTTLE

The Space Shuttle's data processing system has four general purpose computers (GPC) that operate in a redundant set. There are also twenty three multiplexer de-multiplexers (MDM) units aboard the orbiter, sixteen of which are directly connected to the GPCs via redundant shared busses. Each of these MDMs receive commands from guidance navigation and control (GNC) running on the GPC and acquires requested data from sensors attached to it, which is then sent to the GPCs. In addition to their role in multiplexing/demultiplexing data, these MDM units perform analog/digital conversion. Data transfered between the GPC and MDMs is sent in the form of serial digital data.

The GPCs execute redundancy management algorithms that

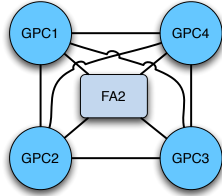


Figure 1: Shuttle Data Processing System (GPCs and FA2)

include a fault detection, isolation, and recovery (FDIR) function. During the launch of shuttle flight Space Transportation System 124 (STS-124), there was reportedly a prelaunch failure of the fault diagnosis software due to a "non-universal I/O error" in the second flight aft (FA2) MDM [3], which is polled by the GPCs as shown in Figure 1. According to [3, 4], the events unfolded as follows:

- A diode failed on the serial multiplexer interface adapter of the FA2 MDM.
- GPC 4 receives erroneous data from FA2. Each node votes and views GPC 4 as providing faulty data. Hence GPC 4 is voted out of the redundant set.
- Three seconds later, GPC 2 also receives erroneous data from FA2. In this case, GPC 2 is voted out of the redundant set.
- In accordance with the Space Shuttle flight rules [22], GPC 2 and GPC 4 are powered down.
- GPC 3 then reads FA2's built-in test equipment and determines that GPC 3 is faulty at which point it too is removed from redundancy set leaving only GPC 1 at which time engineers terminated the work and the problem with FA2 was isolated and the unit replaced.

The above set of events sequentially removed good GPC nodes, but failed to detect and act on the faulty MDM. Based on the analysis reported in [4], it appears the system had a single point of failure. Even though the nodes were connected to the MDM via a shared bus, conditions arose where different nodes obtained different values from MDM FA2.

3. FAULT-TOLERANCE

The terms 'failure', 'error', and 'fault' have technical meanings in the fault-tolerance literature. A *failure* occurs when a system is unable to provide its required functions. An *error* is "that part of the system state which is *liable to lead to subsequent failure*," while a *fault* is "the *adjudged or hypothesized cause* of an error" [20]. For example, a sensor may break due to a fault introduced by overheating. The sensor reading error may then lead to system failure.

We are primarily concerned with architectural-level fault-tolerance [8]. A *fault-tolerant system* is one that continues to provide its required functionality in the presence of faults. A fault-tolerant system must not contain a *single point of failure* such that if that single subsystem fails, the entire system fails (for the faults tolerated). Thus, fault-tolerant systems are often implemented as distributed collections of nodes such that a fault that affects one node or channel will not adversely affect the whole system's functionality.

Faults can be classified according to the hybrid fault model of Thambidurai and Park [28]. Here, we characterize the faults of a node in a distributed system based on the messages other nodes receive from it. The same characterization could be made of channels. First, a node that exhibits the absence of faults is *non-faulty* or *good*. A node is called *benign* or *manifest* if it sends only *benign messages*. Benign messages abstract various sorts of misbehavior that are reliably detected by the transmitter-to-receiver fault-detection mechanisms implemented in the system. For example, a message that suffers a few bit errors may be caught by a cyclic redundancy check. In synchronized systems, nodes that send messages received at unexpected times are also considered to be benign. A node is called *symmetric* if it sends every receiver the same message, but these messages may be arbitrary. A node is called *asymmetric* or *Byzantine* if it sends different messages to different receivers, and at least one of the messages received is not detectably faulty [19]. (Note that the other message may or may not be incorrect.)

We model faulty behavior exclusively within the communication model. This leads to an observational classification of fault effects. The fault effect classification model we employ is derived from the Azadmanesh and Kieckhafer [1] generalization of the Thambidurai and Park [28] hybrid fault model. These fault classifications are from the perspective of the receivers. Each source node is classified according to its worst observed error manifestation. The classification is a function of both the behavior of the node and how that behavior is perceived by a specified collection of observers.

The nodes are classified according to the following definitions:

good All receivers receive correct values.

omissive symmetric All receivers receive either correct values or manifestly incorrect values (including the possibility of no message at all). All receivers observe the same pattern of messages.

omissive asymmetric Some receivers may receive correct messages, while others may receive manifestly incorrect values.

transmissive symmetric All receivers observe the same pattern of messages. Messages may be incorrect.

fully transmissive asymmetric (Byzantine) Receivers may receive arbitrarily different values.

Some protocols are defined to operate correctly under the assumption that the possible fault behaviors are limited to specific subsets of these possible observable fault manifestations. Whenever this is done, there is an obligation to

validate the fault hypotheses, since the various theoretically possible fault manifestations have been observed in both laboratory and deployed systems [11]. Likewise, if a system purports to continue to operate correctly under some bounded number of Byzantine faults, this too must be validated.

In the case study given in Section 2 the nodes were connected to the MDM via a shared bus, yet conditions arose where different nodes obtained different values from MDM FA2. This is consistent with the MDM FA2 failing in a Byzantine fashion sending different values to the GPCs using the topology in Figure 1. Note that the triple-redundancy voting scheme employed in the case of the shuttle would have masked many faults, but not Byzantine faults. The designers may have simply assumed that such Byzantine faults were too unlikely to occur to warrant the additional complexity needed to handle them, yet they appear to have occurred in practice. This illustrates the need for the V&V process to test not only the fault model advertised by the system, but to test the assumptions built into that fault model.

V&V of fault-tolerant systems requires that the test engineer fully exercise the faults purportedly covered by the fault model. In cases where the system's fault model does not accommodate Byzantine faults, it may still be desirable from the V&V perspective to demonstrate that such assumptions actually hold and if not, how a system functions in the presence of such faults. The tests must include faults that are physically possible, but not logically anticipated. Such faults may arise from hardware failures, radiation faults, as well as from traditional inputs. So the V&V engineer must produce a range of inputs, a range of faults, and inject the said faults into the system and observe their effects.

4. FORMAL MODEL

Consider the situation of a *transmitter* node (this could be a sensor as in the shuttle example) that sends data to a number of *receiver* nodes. As we have seen, the transmitter, a receiver, or the interconnect may suffer a fault that causes a receiver to compute a received value that differs from those received by the other receivers. A simple variant of the oral-messages Byzantine protocol [19] can be employed to mask a bounded number of these errors. The protocol that we consider uses *Relay* nodes. The data flow of the protocol is illustrated in Figure 2, where the transmitter sends a message to each of the relay nodes, which, in turn, sends the value they received to each of the end nodes. At the receiver, a majority vote is performed on the values received. If the number of faults are appropriately bounded, then the nodes have achieved consensus.

We built a small PVS model of this protocol, where each node has a core functional component and network interface (NIC) (receiver/sender) components. This architecture is illustrated in Figure 3. The components are connected by FIFO queues. A message sent from the transmitter to the relay nodes is placed in the queue to the transmitter's NIC sender, which places the message in queues connecting it to the receiver NICs at each of the relay nodes. The relay module removes the message from the NIC receiver queue and places it into a queue leading to the NIC sender, which copies the message into the queue of each receiver.

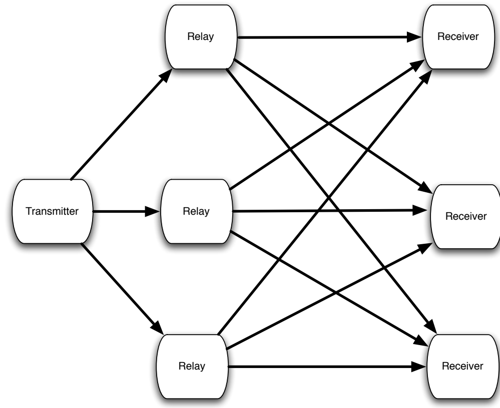


Figure 2: Simple Oral Messages Protocol

We assume a synchronous execution model with the nodes assumed to work in lock-step as if on a global clock. For instance, the transmitter sends its value to its NIC sender at clock tick 1, the value is sent to the relay node's NIC receiver at tick 2, etc.

In the fault model for our system, the transmitter is assumed to be subject to fully transmissive asymmetric errors. The relay nodes are assumed to be more reliable, say due to the use of redundant pairs, but subject to omissive asymmetric faults.

4.1 NIC Receiver

We now consider the PVS model of the NIC receiver in some detail. The NIC receiver is parameterized by the number of inbound communication channels, `maxsize`, and the global time at which the state machine is to execute. The messages are assumed to be of type `Frame`. The state of the NIC receiver is formed from the product of the following:

- `wires` : `below(maxsize)` $\rightarrow$ `fifo[Frame]`. The wires connect the receiver NIC to the sender NIC. As illustrated in Figure 3 each queue in the sequence is connected to a different node.
- `from_nic` : `below(maxsize)` $\rightarrow$ `fifo[Frame]`. This sequence of queues is used to pass the data received to the relay or end receiver state machines.
- `enabled` : `below(maxsize)` $\rightarrow$ `boolean`. If `enabled`(i) is true, then the source node i (the source node attached to `wires`(i)) is assumed to be valid. The value can be false for a number of reasons including messages being dropped or garbled.
- `pc` is the current global clock value.

At this time, we do not model details of buffering, CRC checks etc. Instead, we focus on capturing the fault model at this node. The state machine defining this component has the signature

`NICReciver` $\times$ `AllReceiveActions` $\rightarrow$ `NICReceiver`,

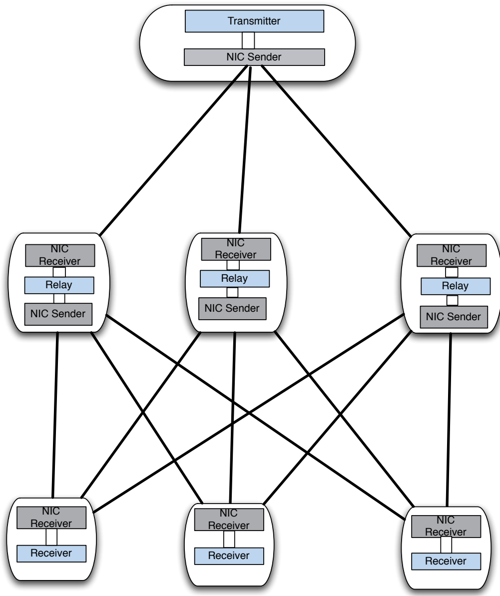


Figure 3: Structure of the Model

where `AllReceiveActions` is defined as

```
AllRecActions:TYPE = below(maxsize) → Actions.
```

The PVS datatype `Actions` defines the allowed faults and is defined as follows:

```
Actions : DATATYPE
  BEGIN
    Good             : Good?
    Garbled          : Garbled?
    Sym(frame:Frame) : Sym?
    Asym(frame:Frame): Asym?
  END Actions.
```

Recall that `wires` hold messages from each incoming channel and faults are not modeled anywhere else in our model so a message is not corrupted or lost in transit. Instead, we use `AllReceiveActions` to inject faults at the NIC receiver. This type acts as a filter so if `AllRecActions`(i) has the value `Good`, the message received is assumed to be good and is relayed on, the value `Garbled` is treated as a benign fault or a dropped message and would be dropped, the value `Sym(frame)` would result in using the value `frame`, with the same value sent to each node, and `Asym(frame)` would result in the value `frame` being placed in the `fom_nic`(i) queue, where each i may get a distinctly different value.

Each of the fault model classifications has an associated PVS type that constrains the values of the filter. For instance good messages are defined by the type:

```
AllGoodRecAction: TYPE =
{f:AllRecAction| ∀ (i:below(maxsize)): Good?(f(i))},
```

omissive asymmetric faults are defined by the type:

```
OmissAsymRecAction : TYPE = {f : AllRecActions |
 ∀ (i:below(maxsize)) : Good?(f(i)) ∨ Garbled?(f(i))}
```

and transmissive asymmetric faults are defined by the type:

```
TransAsymRecAction: TYPE =
{f: AllRecActions | ∀ (i:below(maxsize)):Asym?(f(i))}.
```

The PVS code for updating the `from_nic` component of the `NICReceiver` state is given as follows.

```
λ(i:below(maxsize)):
 CASES a(i) OF
   Good : IF  ¬ empty_fifo?(s`wires(i)) THEN
          enqueue(topof(s`wires(i)),s`from_nic(i))
          ELSE empty_fifo
         ENDIF,
  Garbled : empty_fifo,
  Asym(frame) :  enqueue(frame,s`from_nic(i)),
  Sym(frame)  :  enqueue(frame,s`from_nic(i))
 ENDCASES,
```

which returns a new sequence of queues with the values determined by entries in the action sequence `a`. The state machine also removes the value received from each of the fifo queues comprising `wires`. If message i was garbled, then the corresponding entry in `enabled` is set to false.

Above the state machines is a relational model where the components get connected together and that drives the state machine transitions. In the case of the NIC receiver, the fault model applied to a particular node is chosen. Existential quantification is used to model nondeterministic choice. Ideally we will want the faults to be created in a separate module from the system under test in order to model a test harness.

5. PVS TO JAVA TRANSLATOR

A PVS-to-Java translator [15] has been constructed as part of a collaborative effort between NIA and the Radboud University Nijmegen. The input to our code generator is a declarative specification written in PVS. Since we aim at a wide range of applications, we do not fix the target language. Indeed, the tool first generates code in Why, an intermediary language for program verification [12]. Our current prototype generates Java annotated code from the Why code. In the future, we may implement outputs for other functional and imperative programming languages.

In addition to enabling multi-target generation of code, another benefit of an intermediate language is that transformations and analysis that are independent from the target language can be applied to the intermediate code directly.

Consider the PVS datatype `Actions` defined in the Section 4. Each of the different actions becomes a subclass that extends the class `Actions` and has a constructor that takes the generic class `Frame` as an argument.

```
public class ReceiveAction <Data> {
 public class Actions { public Actions() {}}
 public class Sym extends Actions {
  FrameTh<Data>.Frame frame;
```

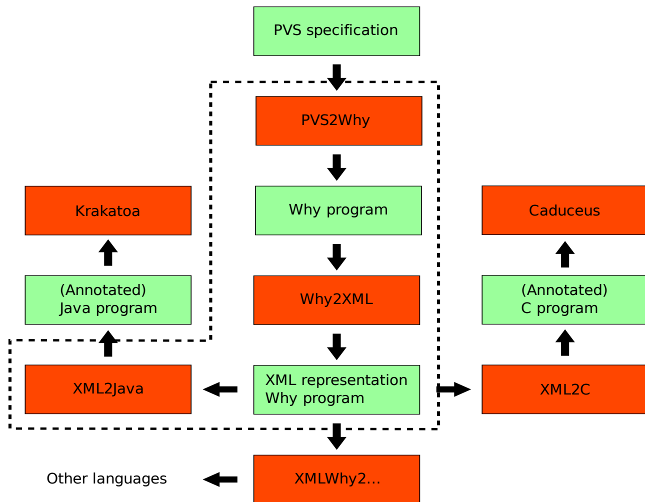


Figure 4: Multi-target generation of verifiable code

```
  public Sym(FrameTh<Data>.Frame frame) {
   this.frame = frame; } }
```

The higher order use of defined functions is facilitated by a special `Lambda` class. This generic abstract class demands that an application function is supplied for each instance.

```
public abstract class Lambda<T1,T2> {
  abstract public T2 apply(T1 obj); }
```

For all defined functions in PVS, a higher order version is generated that satisfies the requirements of the `Lambda` class.

```
public Lambda<Actions,Boolean> SymRecognizer =
 new Lambda<Actions,Boolean>(){
  public Boolean apply(final Actions actions){
   return SymRecognizer(actions);}};}
```

All functions are translated into currified syntax. This way it is possible to translate all higher order uses of functions, including partial application, into working Java programs.

In its current phase of development, the translator can only translate PVS specifications that are written in functional style, in particular, as a state machine. Relational specifications, which typically model nondeterministic behavior, must be hand coded. The existential quantification in the specification, which is used to nondeterministically generate actions, gets replaced by hooks into the Symbolic Java PathFinder tool.

6. SYMBOLIC (JAVA) PATHFINDER

For test case generation, we will use Symbolic (Java) PathFinder (SPF) [9], a symbolic execution framework built on top of the Java PathFinder model checker [23]. SPF combines symbolic execution and constraint solving techniques for the automated generation of test cases that achieve high coverage. Symbolic PathFinder implements a symbolic execution framework for Java byte-code. It can handle mixed integer and real inputs, input data structures and strings, as well as multi-threading and input pre-conditions.

Symbolic execution [18] is a well-known program analysis technique that uses symbolic values instead of actual data as inputs and symbolic expressions to represent the values of program variables. As a result, the outputs computed by a program are expressed as a function of the symbolic inputs. The state of a symbolically executed program includes the (symbolic) values of program variables, a path condition (PC), and a program counter. The path condition is a boolean formula over the symbolic inputs, encoding the constraints which the inputs must satisfy in order for an execution to follow the particular associated path. These conditions can be solved (using off-the-shelf constraint solvers) to generate test cases (test input and expected output pairs) guaranteed to exercise the analyzed code.

Symbolic PathFinder implements a non-standard interpretor for byte-codes on top of JPF. The symbolic information is stored in attributes associated with the program data and it is propagated on demand during symbolic execution. The analysis engine of JPF is used to systematically generate and explore the symbolic execution tree of the program. JPF is also used to systematically analyze thread interleavings and any other forms of non-determinism that might be present in the code; furthermore JPF is used to check properties of the code during symbolic execution. Off-the-shelf constraint solvers/decision procedures are used to solve mixed integer and real constraints. We handle loops by putting a bound on the model-checker search depth and/or on the number of constraints in the path conditions.

By default, Symbolic PathFinder generates vectors of test cases, each test case representing input-output vector pairs. In order to test reactive systems, such as the fault tolerance protocols that we are studying here, we have extended Symbolic PathFinder to also generate test sequences (i.e., sequences of test vectors) that are guaranteed to cover states or transitions in the models (other coverages such as condition, or user-defined are also possible). This works by instructing Symbolic PathFinder to generate and explore all the possible test sequences up to some user pre-specified depth (or until the desired coverage is achieved) and to use symbolic, rather than concrete, values for the input parameters.

7. PRELIMINARY RESULTS: TEST CASE GENERATION

We have begun applying Symbolic PathFinder to the Java code generated from the PVS specifications for the case study described in Section 4. Each of the PVS models for the components of the system (wires, senders, receivers, relays, and so on) was translated into a Java class through the process described in Section 5. We applied SPF to these classes combined with the hand-written driver code (also described in Section 5).

Both the connections among the components and the model's execution policy are implemented in the driver code. The Java code that was automatically derived from the PVS specifications does not assume any particular inter-component connections or execution policy. Since this case study initially assumes a synchronous execution policy with a global clock, our first implementation of a driver is a simple single-threaded model which can be outlined as:

```
... wire components together ...

for (int pc=0; pc<maxPc; pc++) {
  sender1.step();
  nicSender1.step();
  nicReceiver1.step();
  ...   }
```

To perform test case generation, we identified the *inputs* to the protocols as being the different types of faults that can be injected on the receiver side to test the fault tolerance behavior. In the PVS model, the four possible component fault types (`Good`, `Garbled`, `Symmetric`, `Asymmetric`) are introduced into the NIC Receiver through the `Actions` PVS datatype (see Section 4.1). However, our Java code simplifies this by implementing the four fault types as simple integers in the range `0,..,3`. This is only for convenience — it does not materially affect the model.

As a second simplification for this initial trial, we modeled only a single sender wired to two receivers. Future runs will expand the model to cover the full case study shown in Figure 3.

In order to generate test cases with SPF, we needed to annotate the Java code so that SPF knows that the integer `Actions` are the symbolic variables for which path conditions must be derived. This is easily done. The original driver code which sets up the array of actions (i.e. possible faults):

```
int[] actions = new int[maxsize];
for (int i = 0; i < maxsize; i++)
  actions[i] = ...some fault type in 0,1,2,3...;
```

is modified to tell SPF that the integers in `actions[]` are to be treated symbolically:

```
int[] actions = new int[maxsize];
for (int i = 0; i < maxsize; i++)
  String symVarName = "rcvr-" + (rcvrID++) "-error";
  actions[i] = Debug.getSymbolicInt(0,3,symVarName);
```

`Debug.getSymbolicInt(min, max, name)` is a utility method in SPF which tells the model checker to generate a symbolic integer named `name` whose range is `min`...`max`. This is the only annotation needed to apply SPF.

Running SPF then produces the test inputs from the path conditions:

```
Constraint 1:
rcvr-1-error == Good &&
rcvr-0-error == Good

Constraint 2:
rcvr-1-error == Garbled &&
rcvr-0-error == Good

...

Constraint 15:
rcvr-1-error == Symmetric &&
rcvr-0-error == Asymmetric

Constraint 16:
rcvr-1-error == Asymmetric &&
rcvr-0-error == Asymmetric
```

We then ran the Java code with these sixteen tests and measured the coverage. In some Java classes the coverage was less than 100%. Examining the code that was not executed showed that most of statements were in code that is not needed for the simple example used for this initial trial. A few unexecuted lines are significant and are the result of problems in the handwritten driver code. These problems will be addressed in future our future work.

8. RELATED WORK

Almost all of the major theorem provers provide some form of code generation their specification language. For instance, theorem prover Isabelle/HOL even provides two code generators [2, 16], ACL2's [17] specification language *is* a subset of Common Lisp, and Coq [5] has a generator [21] that extracts lambda terms and translates them in either Haskell or OCaml. Unlike the generator we use in this paper, these languages are all functional programming languages.

While there has been a lot of work on specification-based testing and test case generation [10, 13, 14], there has been little work focusing on bridging the gap between theorem proving and testing. The HOL-TestGen system [7] generates unit tests from Isabelle [24] specifications. The literature currently focuses on generating tests for common libraries. Sewell *et al.* have constructed a tool that uses HOL specifications as an oracle for testing protocols [6], but their focus is not on test case generation. An experiment in using PVS strategies to create random test cases directly from PVS specifications is reported in [25].

The work presented here is also related to the use of formal methods (including theorem proving and model checking) for analyzing fault tolerance of circuits and systems [26, 27]. In contrast to these works, our goal is to leverage the effort of building and formally verifying models of such systems into testing actual implementations.

9. CONCLUSION AND FUTURE WORK

Formal methods are increasingly accepted in the fault-tolerant systems community as a means to analyze the correctness of a design under the assumption of a well specified fault model. Yet testing must be employed to validate an executable against a model. Furthermore, testing should explore whether assumptions built into the fault model do indeed hold. We present a methodology whereby a PVS formal model drives the creation of a Java executable prototype that can be used as reference implementation. Test cases can then be generated by applying static analysis techniques to the prototype implementation. The PVS-to-Java translator and the Symbolic PathFinder tools enable this process by automating much of the task.

The work reported in this paper is still preliminary and much work remains to be done. The PVS-to-Java translator is still evolving as features are added that will allow us to automatically translate more of our model. SPF is similarly evolving.

For instance, the ability to generate tests from the abstract datatype `Actions` is under development.

The results reported in Section 7 reflect experiments intended to test the feasibility of the methodology and to drive what features need to be added to our tools. Our next milestone is to be able to exercise the full protocol seen in Section 3. Furthermore, we need to extend the generation of test cases to also include the expected output (e.g. various observable protocol states) as our goal is to use these test cases to test actual implementations. The protocol under consideration is very basic. Once we have mastered the process for this example, we expect to focus on more sophisticated protocols such as fault-tolerant distributed clock synchronization.

Acknowledgments

We thank Eric Cooper, Mike Lowry and César Muñoz for their comments. This work was partially supported by NASA Cooperative Agreement NCC1-02043.

10. REFERENCES

[1] M. H. Azadmanesh and R. M. Kieckhafer. Exploiting omissive faults in synchronous approximate agreement. *IEEE Transactions on Computers*, 49(10):1031–1042, 2000.

[2] S. Berghofer and T. Nipkow. Executing higher order logic. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs (TYPES 2000)*, volume 2277 of *LNCS*, pages 24–40. Springer, 2002.

[3] C. Bergin. Faulty MDM removed. NASA Spaceflight.com, May 18 2008. Available at `http://www.nasaspaceflight.com/2008/05/sts-124-frr-debate-outstanding-issues` -faulty-mdm-removed/. (Downloaded Nov 28, 2008).

[4] C. Bergin. STS-126: Super smooth endeavor easing through the countdown to l-1. NASA Spaceflight.com, November 13 2008. Available at `http://www.nasaspaceflight.com/2008/11/sts-126-endeavour-easing-through-countdown/`. (Downloaded Feb 3,2009).

[5] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[6] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and Sockets. In *Proceedings of SIGCOMM 2005: ACM Conference on Computer Communications (Philadelphia),* published as Vol. 35, No. 4 of *Computer Communication Review*, pages 265–276, Aug. 2005.

[7] A. Brucker and B. Wolff. Test-sequence generation with hol-testgen - with an application to firewall testing. In B. Meyer and Y. Gurevich, editors, *Tests and Proofs*, Lecture Notes in Computer Science 4454. Springer-Verlag, 2007.

[8] R. W. Butler. A primer on architectural level fault tolerance. Technical Report NASA/TM-2008-215108, NASA Langley Research Center, 2008.

[9] C. Pasareanu and P. Mehlitz and D. Bushnell and K. Gundy-Burlet and M. Lowry and S. Person and M. Pape. Combining Unit-Level Symbolic Execution and System-Level Concrete Execution for Testing NASA Software. In *International Symposium on Software Testing and Analysis*, pages 15–26. ACM Press, 2008.

[10] J. Chang and D. J. Richardson. Structural specification-based testing: Automated support and experimental evaluation. In *Proceedings FSE'99*, pages 285–302, Sept. 1999.

[11] K. Driscoll, B. Hall, H. Sivencrona, and P. Zumsteg. Byzantine fault tolerance, from theory to reality. In *The 22nd International Conference on Computer Safety, Reliability and Security SAFECOMP*, Lecture Notes in Computer Science, pages 235–248. Springer, September 2003.

[12] J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, UniversitÃľ Paris Sud, Mar. 2003.

[13] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proceedings of the 7th European engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineer ing*, pages 146–162. Springer-Verlag, 1999.

[14] J. Goodenough and S. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, June 1975.

[15] A. E. Goodloe, L. Lensink, and C. Muñoz. From verified specifications to verifiable software. Technical report, National Institute of Aerospace, 2008.

[16] F. Haftmann and T. A code generator framework for Isabelle/HOL. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics: Emerging Trends Proceedings*, number 364/07, 08 2007.

[17] M. Kaufmann, J. Moore, and P. Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[18] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[19] Lamport, Shostak, and Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, July 1982. Available at `http://citeseer.nj.nec.com/lamport82byzantine.html`.

[20] J.-C. Laprie. Dependability—its attributes, impairments and means. In B. Randell, J.-C. Laprie, H. Kopetz, and B. Littlewood, editors, *Predictability Dependable Computing Systems*, ESPRIT Basic Research Series, pages 3–24. Springer, 1995.

[21] P. Letouzey. A New Extraction for Coq. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002*, volume 2646 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[22] NASA - Johnson Flight Center. Space shuttle operational flight rules Volume A, A7-104, June 2002. Available from `http://www.jsc.nasa.gov` (Downloaded Nov 28, 2008).

[23] NASA Ames. Java PathFinder Version 3.1.1 User

Guide.

[24] T. Nipkow, L. Paulson, and W. Wenzel. *Isabelle HOL - A proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science 2283. Springer Verlag, 2002.

[25] S. Owre. Random testing in pvs. In *Workshop on Automated Formal Methods*, 2006.

[26] J. Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. *IEEE Trans. Softw. Eng.*, 25(5):651–660, 1999.

[27] S. A. Seshia, W. Li, and S. Mitra. Verification-guided soft error resilience. In *Proc. Design Automation and Test in Europe (DATE)*, April 2007.

[28] P. Thambidurai and Y.-K. Park. Interactive consistency with multiple failure modes. In *7th Reliable Distributed Systems Symposium*, pages 93–100, October 1988.

SAL, Kodkod, and BDDs for Validation of B Models

Lessons and Outlook

Daniel Plagge Michael Leuschel

Lehrstuhl Softwaretechnik und Programmiersprachen
Institut für Informatik
University of Düsseldorf
{plagge,leuschel}@cs.uni-duesseldorf.de

Ilya Lopatkin Alexei Iliasov
Alexander Romanovsky

School of Computing Science
Newcastle University
{Ilya.Lopatkin, Alexei.Iliasov,
Alexander.Romanovsky}@newcastle.ac.uk

Abstract

ProB is a model checker for high-level B and Event-B models based on constraint-solving. In this paper we investigate alternate approaches for validating high-level B models using alternative techniques and tools based on using BDDs, SAT-solving and SMT-solving. In particular, we examine whether ProB can be complemented or even supplanted by using one of the tools BDDBDDB, Kodkod or SAL.

Categories and Subject Descriptors I.2.3 [*Artificial Intelligence*]: Deduction and Theorem Proving—Logic Programming; D.2.4 [*Software Engineering*]: Software/Program Verification—Model Checking; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification; Specification techniques

General Terms Languages; Verification

Keywords Logic Programming; formal methods; model checking; verification; animation.[1]

1. Introduction

Event-B is a formal method for state-based system modelling and analysis evolved from the B-method [1]. The B-method itself is derived from Z and based upon predicate logic with set theory and arithmetic, and provides a wide array of sophisticated data structures (sets, sequences, relations, higher-order functions) and operations on them (set union, difference, function composition to name but a few).

Event-B has a tool support in a form of the Rodin Platform [2], which is extensible with plugins. The platform supports the mechanisms, essential for rigorous model development among which model checking (exemplified by ProB) plays an important role in ensuring the model correctness and understanding the system behaviour.

ProB [14, 16] is an animator for B and Event-B built in Prolog using constraint-solving technology. It incorporates optimisations such as symmetry reduction and has been successfully applied to several industrial case studies.

Still, sometimes the performance of the tool is suboptimal, especially in the context of complicated properties over unknown variables, which is the reason we have investigated alternate approaches for animating or model checking B specifications. More precisely, we investigate whether model checking of B/Event-B could by done by mapping to the input languages for other tools making use of either SAT/SMT solving techniques or of BDDs. If so, what is the performance difference compared to ProB's current constraint solving approach? Can the constraint solving approach be combined with the alternate approaches? These are the questions we try to answer in this paper. In Section 2 we provide some background information about ProB and its constraint solving approach. In Section 3 we elaborate on our experience in trying to use the BDDBDDB package [28], which provides a Datalog/relational interface to BDDs and has been successfully used for scalable static analysis of imperative programs. In Section 4 we present our experience and first results in mapping B to Kodkod [26], a high-level interface to SAT-solvers used by Alloy [12]. In Section 5, we turn our investigation to SAL [24], based on the SMT-solver Yices. We conclude with a discussion about related and future work in Section 7.

2. ProB

ProB [14, 16] is an animator and model checker for the B-method based on the constraint solving paradigm. Constraint-solving is used to find solutions for B's predicates. As far as model checking technology is concerned, ProB is an explicit state model checker with symmetry reduction [15, 27, 19]. While the constraint solving part of ProB is developed in Prolog, the new LTL model checking engine [20] is encoded in C.

Some of the distinguishing features of ProB are

- the support for almost full syntax of B, integration into Atelier B and Rodin,
- the support for Z [23] and Event-B, building on the same kernel and interpreter as for "classical" B,
- the support for other formalisms such as CSP [18] via custom Prolog interpreters which can be linked with the B interpreter [7].

However, in this paper we will concentrate on B and Event-B. ProB uses a custom built constraint solver over the datatypes of B. The base types of B are booleans, integers and user-defined base

[1] This research is partially supported by the EU funded FP7 project 214158: DEPLOY (Industrial deployment of advanced system engineering methods for high productivity and dependability).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AFM'09, June, 2009, Grenoble, France.

Copyright © 2009 ACM 978-1-60558-117-0/08/07. . . $5.00

sets, while the composed types can be constructed using cartesian product and the power set constructions. As such, B caters for sets, relations, functions and sequences and provides many custom operators on these datatypes (e.g., computing the inverse of a relation, composing relations, ...). Note that the relations and functions can be higher-order (and often are).

The performance of PROB is often good for animation and it has been successfully applied to a variety of industrial specifications. Recently, PROB has been extended to also deal with very large sets and relations (with tens of thousands of elements or more) and it is planned that PROB will be used by Siemens in production in 2009 for validating assumptions about rail network topologies [17].

In some real life scenarios, PROB can actually be more efficient than, e.g., Spin or SMV working on equivalent lower-level models (see [13] or or [11]). Still, there are many scenarios where the performance is not (yet) adequate.

For example, PROB's performance can be disappointing when values for variables have to be found which satisfy complicated predicates, and those constraints allow little deterministic constraint propagation to occur (see, e.g., the example in Section 4.3). For example, finding values for the constants of a B model which satisfy complicated predicates can be very challenging. Another scenario with similar characteristics is the use of PROB as a disprover [4] for proof obligations: here one wants to find values which make the hypotheses of the proof true but the consequent false.

In this paper we investigate whether PROB can be complemented by other technologies in order to improve its performance. We also study whether the entire constraint solving engine could be replaced, if other technologies turn out to be universally superior.

3. BDDs via Datalog

Symbolic model checking with binary decision diagrams (BDDs) has become very popular since the very successful applications on hardware models [6]. We investigated, if and how we could use this approach for Event-B or B models.

BDDBDDB [28] offers the user a Datalog-like language that aims to support program analysis. It uses BDDs to represent relations and compute queries on these relations. We wanted to use the tool to find states that violate the invariant of a model, using Datalog queries that follow the schema

```
check(S) :- init(I),do_events(I,S),inv_violated(S).
do_events(A,A).
do_events(A,B) :- step(A,C),do_events(C,B).
step(A,B) :- event_X(A,B).
step(A,B) :- event_Y(A,B).
```

`check(S)` should return a reachable state that violates the invariant. To find such a state, we start in an initial state `I`, do zero or more operations from `I` to `S` via `do_events` and check if the resulting state violates the invariant with `inv_violated`. `do_events(A,B)` is specified by doing either zero steps (`A` and `B` are the same) or doing one step to an intermediate state `C` and continuing recursively. `step` again models the transition between two states by an event, here e.g. `event_X` and `event_Y`.

What can not be seen in the query above is how an initial state, a state that violates the invariant or an event is specified. To do that, one has to represent a state of the model as a bit-vector and events have to be implemented as relations between two of those bit-vectors. These relations have to be constructed by creating BDDs directly with the underlying BDD library (JavaBDD) and storing them into a file.

If we take e.g. a model that contains two integers a and b, and an event with the action $a := a + b$, we have to define a boolean formula that specifies for every bit of a in the new state how it correlates with the bits of a and b in the original state and for all other bits that they stay the same.

Soon after starting experimenting with BDDBDDB it became apparent that due to the lack of more abstract data types than bit vectors, the complexity of a direct translation from B to BDDBDDB was too high, even for small models. So we abandoned this approach, especially as there are other tools like SAL or Kodkod that give us the possibility to use symbolic approaches but offer a more powerful interface to define the models.

4. SAT Solving via Kodkod

Kodkod [26] is a constraint solver for first order logic that offers an extensive set of operations on relations. It uses an underlying SAT-solver (like *minisat*) to find solutions to a given problem.

It seems to be much more suitable for our purpose than the previously mentioned low-level approach using BDDs, because sets and relations (Kodkod considers sets as unary relations) are heavily used in B specifications. To use Kodkod, one basically has to provide four things to find solutions for a problem:

- An *universe* of atoms.
- n-ary *relations* between the atoms.
- A predicate, called *formula*, is constructed as an abstract syntax tree and can refer to previously defined relations.
- *Bounds* on the relations define which atoms can be in each relation.

Kodkod then provides possible instances for the relations.

Kodkod itself does not define an input language, but comes as a Java library and the user defines the components of the problem via the API. We used this library to implement a component of PROB that runs in a separate process. We do not replace a whole B specification by a kodkod problem description but we rather replace single predicates. This allows us to mix Kodkod and PROB's constraint solving technique, which is particular useful if components of the specification are not translatable.

4.1 Translation from B to Kodkod

In a first approach, we restricted ourselves to expressions that used only deferred sets or enumerated sets as data types. To translate a B predicate into a Kodkod problem, we do the following:

First, we construct the atoms of Kodkod's universe by creating an atom for each element of an enumerated set. Deferred sets can be treated the same way, because PROB assigns a fixed finite cardinality to each deferred set.

Then we can translate the given (i.e. enumerated or deferred) set into an unary relation which contains exactly all atoms that belong to its elements. The information that the relation contains exactly those atoms is specified as a bound on the relation. We translate each element of an enumerated set to an unary singleton relation which contains exactly the associated atom.

For every variable (or constant) that is referenced in the predicate we create a Kodkod relation. If the variable's type is a set or relation, the translation is straight-forward, if the variable's type is an element of a deferred or enumerated set, we have to add the predicate that the relation is a singleton.

B expressions often have a direct Kodkod counterpart: E.g. the cartesian product, set union, intersection and difference, reverse and closure of a relation, the relational image. Some expressions can be translated using other existing constructs. E.g. the domain restriction $S \lhd R$ with $S \subseteq A$ and $R \in A \leftrightarrow B$ can be rewritten into $S \lhd R = R \cap (S \times B)$, that again can be translated directly. Identifiers are translated to references to the according relations, as described above. Also many B predicates like conjunction, disjunc-

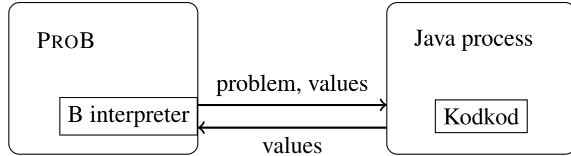


Figure 1. PROB and Kodkod

tion, negation, universal and existential quantification, set membership, subsets, etc. can be directly translated.

Fig. 3 shows an example of a predicate in B syntax and the corresponding Kodkod formula. The predicate is part of a specification of a control flow analysis and we use it below for a small performance comparision between Kodkod and PROB. The formula is taken from the output of the `toString` method of the corresponding Java object and only slightly modified to improve readability. One can see that it's very similar to the original B predicate, the additional `one lentry` in the beginning states that `lentry` is a singleton set. `&&` is the logical conjunction, `.` the relational image, `&` the intersection, `~` the inverse and `->` the cartesian product.

Limitations of the translation Currently the data type of each expression that we translate must be an element of a given set, a subset of a given set, or a relation between those types. Our approach does *not* support types like sets of sets, as those cannot be translated to Kodkod easily.

Another limitation is that we currently do not consider integers. Kodkod has limited support for integers, one can specify integers and sets of integers and use basic operations like addition or multiplication on them. Internally, Kodkod maps each integer to an atom (at least if a set of integers is used) and uses a bit encoding with a fixed number of bits for the integer operations. Integer overflows are silently ignored. We are currently working on a version which first does a static analysis on the B predicate to determine the possible intervals of each expression. This is needed to pass a maximum bit width to Kodkod that guarantees the integer operations to be correct. Additionally, we need the intervals to create an atom for each possible integer if a set of integers is used in an expression.

4.2 Interaction between PROB and Kodkod

To make use of Kodkod's features in PROB, a predicate in the internal syntax tree is replaced by a special syntax element that describes a Kodkod representation of the predicate. When the interpreter encounters this element the first time, the description of the problem is sent to a separate Java process (see Fig. 1). After this initialisation phase, each time the interpreter evaluates the predicate, the currently known values of used variables are sent to the process, which in turn returns possible values (calculated by Kodkod) for the remaining variables of the predicate.

In the example above, PROB would send the value of $succ$, and the Java process returns the 7 possible values for L and $lentry$.

4.3 Performance comparison

For now, we do not have an exhaustive performance comparison between PROB and Kodkod. Evaluating two extreme examples suggests that depending on the problem, either approach can show its strength.

Kodkod and large relations Kodkod does not seem to scale well when encountering large relations. This has only been relevant for certain applications of PROB, such as the property verification on real data [17]. The log-log plot in Figure 2 contains a small experiment where the performance of the set-difference operation is analysed. PROB scales linearly, while Kodkod exhibits an exponential growth (slope of the Kodkod curve > 1).

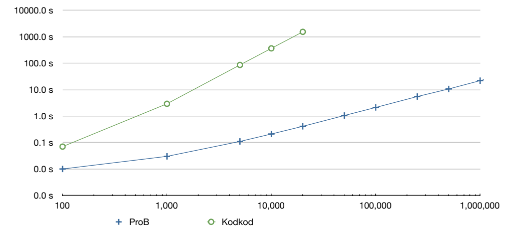


Figure 2. Performance of PROB vs Kodkod on large sets

$$Blocks = \{b_1, b_2, b_3, b_4, b_5, b_6, b_{entry}, b_{exit}\}$$
$$succs = \{b_{entry} \mapsto b_1, b_1 \mapsto b_2, b_2 \mapsto b_3, b_3 \mapsto b_3, b_3 \mapsto b_4, b_4 \mapsto b_2, b_4 \mapsto b_5, b_5 \mapsto b_6, b_6 \mapsto b_6, b_6 \mapsto b_{exit}\}$$

The predicate in B syntax

$$lentry \in L \wedge succs^{-1}[L \setminus \{lentry\}] \subseteq L \wedge \forall l.(l \in L \Rightarrow lentry \in (L \lhd succs \rhd L)^{+}[\{l\}])$$

and encoded in Kodkod

```
one lentry && lentry in L &&
((L-lentry) . ~succs) in L &&
all l: one Blocks | (l in L =>
  lentry in (l.^(((L->Blocks)&succs)&(Blocks->L))))
```

Figure 3. Finding loops in a control flow graph

Finding loops in a control flow graph For a certain class of problems, Kodkod is much faster. The problem given in Fig. 3 is encoded in a B-machine by defining the basic blocks $Blocks$ as an enumerated set, the relation $succs$ as a constant and the predicate as a precondition for an operation with two parameters L and $lentry$.

PROB needs 683 sec to find all 7 solutions without Kodkod-support. When the predicate is replaced by a call to the Kodkod-process, the computation time reduces to 10 ms.

4.4 Conclusion and ongoing work

For certain problems, the use of a SAT solver via Kodkod seems very promising. We do not see that this approach might replace the constraint solving mechanism of PROB, because there still will be constructs in specifications that are very hard to translate.

The lack of support for integers turned out to be a hurdle for trying out more examples. So we are currently working on supporting integers and will then continue the evaluation of this approach.

The possibility to mix predicates that are handled by Kodkod or PROB's constraint solving algorithm in the interpreter has the advantage that we can use Kodkod without having to drop the support for part of the specification language. However, for making symbolic techniques like bounded model checking available, we must be able to translate the whole specification, because PROB's interpreter just supports explicit model checking. We are currently thinking about ways to circumvent this restriction by using predicate abstraction [8].

5. SAL

SAL [24] is a model-checking framework combining a range of tools for reasoning about information systems, in particular concurrent systems, in a rigorous manner. The core of SAL is a language

for constructing system specifications in a compositional way. The SAL tool suite includes a state of the art symbolic (BDD-based) and bounded (SAT-based) model checkers.

The overall aim of our work is to investigate the potential applications of SAL in a combination with the Rodin platform, a development tool for the Event-B formalism. Unlike SAL, Rodin relies mostly on theorem proving for model analysis. We are looking for a way of complementing the Rodin platform with a plugin that would automate some of the more difficult tasks such as liveness, deadlock freeness and reachability analysis.

5.1 Event-B to SAL translation

In this section we describe our ongoing work on translating Event-B models into the input language of SAL. We report the results on initial experiments on verifying Event-B models in the SAL framework. The benchmark for our efforts is PROB[14, 16]. Since PROB is Prolog-based, we had started with an expectation of achieving some performance advantage for a considerable subset of problems.

Table 1 and Figure 4 present the comparative performance of the PROB Event-B model checker and SAL [2] run on the result of transforming the same model into the input language of SAL. The first model is a synthetic benchmark based on bubble sort algorithm. In this model a single event swaps neighbouring elements of an array if they are in a wrong order. The other four models are the examples bundled with PROB distribution. These demonstrate the translation and the performance of some of the most "inconvenient" parts of Event-B syntax for SAL: sets, functions, relations and operators on them such as union, intersection, cardinality and etc.

In the comparison tables "SAL run" stands for the time of iterating through the state space, and "SAL total" is the total time including generation of model checker. On the charts we show PROB and total SAL times with values connected by lines where there is an evident dependency between the size of state space and corresponding timings. During model checking the Club specification we were changing several parameters, and each of them had its effect on the size of state space and total model checking time. Thus we show results for this model as a set of non-connected points. All figures are meaningful for comparative analysis only: they could change substantially depending on the operating system, compiler distribution, and, obviously, the performance of a machine on which tests are run.

These results are obtained with default settings in both model checkers. By enabling hash or nauty *symmetries*, one is able to obtain much better results in PROB (see also [13]) , although this requires some understanding on when these options should be enabled. In some cases, symmetry reduction may slow down the process but mostly it reduces the model checking time. For example, with hash symmetry enabled, PROB can model check the Scheduler specification with parameter set to 5 in just 0.25s compared to 1.7s with default settings (see Table 2). Also, for Life with set size 20, PROB only generates 232 states with symmetry and the model checking time goes down to 2.2 seconds. Finally, for Club with MaxInt = 20 and set size 4, the number of states goes down to 682 from 3597, and the model checking time goes down to roughly 6 seconds.

Reasoning on timings we obtained during our experiments, we drew preliminary conclusions about efficiency of SAL model checker on our models:

- SAL verification stage alone can be more than 10x up to 1000x faster than PROB (without symmetry) for a large class of models;
- even in the case of enabled symmetry in PROB, SAL shows either better or comparable performance;
- the bottleneck of SAL model checking performance is in the pre-verification analysis and checker generation stages. In particular, unfolding quantifiers may take $\sim$ 95% of generation time;
- SAL model checking time strongly depends on the complexity of theorems, complex computations in theorems dramatically reduce the overall performance.

In these models, we used a classical representation of sets in predicates adopted from [25]. Let us look at an excerpt from a B model which uses sets:

Sort				
Parameters	*States*	PROB	*SAL run*	*Sal total*
array[3] of 1..10	1000	18 sec	0.08 sec	0.2 sec
array[4] of 1..10	10000	5 min 41 sec	0.1 sec	0.3 sec
array[6] of 1..10	1000000	$>$ 30 min	13 sec	14 sec
array[6] of 1..12	2985984	$>$ 30 min	29.5 sec	30.7 sec
Life				
Parameters	*States*	PROB	*SAL run*	*Sal total*
set size = 5	243	0.9 sec	0.06 sec	0.28 sec
set size = 7	2187	11.8 sec	0.06 sec	0.42 sec
set size = 8	6561	72.5 sec	0.06 sec	0.8 sec
set size = 20	$> 10^9$	$>$ 20 min	0.28 sec	2.7 sec
Max				
Parameters	*States*	PROB	*SAL run*	*Sal total*
MaxInt = 7	256	1.4 sec	0.06 sec	0.22 sec
MaxInt = 8	512	3.3 sec	0.05 sec	0.25 sec
MaxInt = 9	1024	7.8 sec	0.08 sec	0.26 sec
MaxInt = 10	2048	20.7 sec	0.08 sec	0.28 sec
MaxInt = 11	4096	61 sec	0.09 sec	0.25 sec
MaxInt = 12	8192	3 min 18 sec	0.2 sec	0.6 sec
MaxInt = 30	$> 10^9$	$>$ 30 min	0.4 sec	2 sec
MaxInt = 50	$> 10^{15}$	$>$ 30 min	2.2 sec	7.4 sec
Scheduler				
Parameters	*States*	PROB	*SAL run*	*Sal total*
set size = 3	35	0.24 sec	0.05 sec	0.47 sec
set size = 4	124	0.7 sec	0.1 sec	1.8 sec
set size = 5	437	1.7 sec	0.1 sec	55 sec
Club				
Parameters	*States*	PROB	*SAL run*	*Sal total*
capacity = 1..2 set size = 3 MaxInt = 10	368	1.8 sec	0.03 sec	0.6 sec
capacity = 1..2 set size = 4 MaxInt = 10	958	4.6 sec	0.03 sec	5 sec
capacity = 1..2 set size = 4 MaxInt = 30	3358	11.8 sec	0.03 sec	5.5 sec
capacity = 1..2 set size = 4 MaxInt = 50	5758	30.3 sec	0.05 sec	6 sec
capacity = 1..2 set size = 4 MaxInt = 10000	119758	$>$ 30 min	0.05 sec	6 sec
capacity = 1..2 set size = 5 MaxInt = 10	2408	9.5 sec	0.05 sec	4 min 29 sec
capacity = 1..3 set size = 4 MaxInt = 20	3597	14.2 sec	0.03 sec	5.4 sec

Table 1. Comparison on Event-B and SAL specifications

[2] All timings are obtained using *sal-smc* model checker

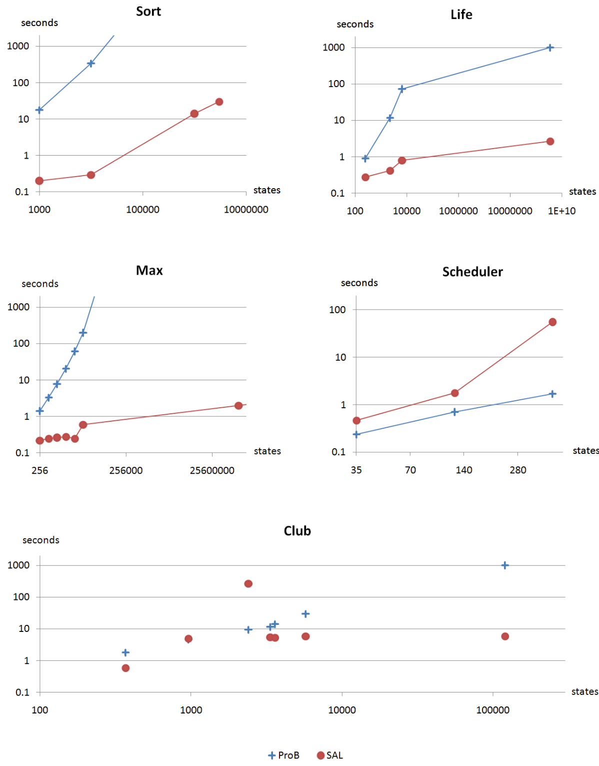


Figure 4. Comparison on Event-B and SAL specifications

```
SETS s ...
CONSTANTS total ...
VARIABLES a ...
INVARIANT a⊂s & card(a)≤total ...
OPERATIONS
 add(b) = WHEN b∈s & b∉a THEN a:=a∪{b} END;
 ...
```

The corresponding SAL specification is

```
modelname: CONTEXT =
BEGIN
 ntype: TYPE = {n: prob!NAT1 | n <= prob!MaxSetSize};
 s: CONTEXT = set{ntype;};
 main: MODULE =
 BEGIN
   LOCAL a: s!Set ...
   TRANSITION [
   ([](b:ntype): add:
     NOT s!contains(a, b) --> a' = s!union(a, b))...
 END
 th: theorem main ⊢
      G(EXISTS(n: prob!SizeType):
           s!size(a, n) AND n<=total);
END
```

And the implementation of cardinality test is

```
size(s: Set, n: natural): boolean =
 (n=0 and equals(s, Empty)) or
  (n>0 and exists (f: [[1..n] -> T]) :
   (forall (x1, x2: [1..n]): f(x1)=f(x2) => x1=x2) and
      (forall (y: T): s(y) <=>
           (exists (x: [1..n]) : f(x)=y)));
```

Such calculation of cardinality involves an excessive use of quantifiers. In [9] a brute-force approach is proposed which iterates through a function from a set type to boolean. We believe that this can still be improved upon. Since SAL natively supports arrays, we encoded sets as arrays of boolean type with an index being mapped to values of a set type. Essentially, a set (and its derivatives such as relation and function) is represented as a characteristic function. The approach brings a number of advantages. For instance, cardinality calculation is realised with a relatively efficient recursive function:

```
sizesofar(s: Set, n: SetArrayType): prob!SizeType =
 if s[n] then 1 else 0 endif +
 if n>1 then sizesofar(s, n-1) else 0 endif;
size(s: Set): prob!SizeType = sizesofar(s, prob!MaxSetSize);
```

As a further optimisation technique, for each set we introduce an auxiliary variable storing the current set size. This variable is updated each time the set is changed.

With our new style of set translation, the example above is transformed into the following:

```
modelname: CONTEXT =
BEGIN
 s: CONTEXT = set;
 main: MODULE =
 BEGIN
   LOCAL a: s!Set, a_size: prob!SizeType ...
   INITIALIZATION ...; n = set!size(a); ...
   TRANSITION [
   ([](b:set!SetArrayType): add:
     NOT s!contains(a, b) -->
       a' = s!union(a, b); a_size' = set!size(a');)...
 END
 th: theorem main ⊢ G(a_size <= total);
END
```

With this approach an invariant (SAL safety theorem) does not involve the heavy calculation of a set size. Unsurprisingly, this results in a significant performance benefit for the models operating on sets. We present the comparison of these timings in Table 2 and Figure 5.

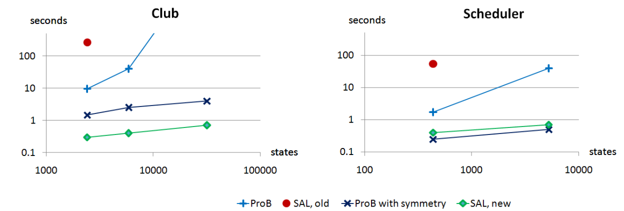


Figure 5. The comparative results for the efficient set translation (Sal, new)

5.2 Ongoing work on the plugin

Our ongoing work is focusing on incorporating the SAL model checker into the Rodin platform. A number of steps are being performed to achieve this. We are aiming at developing a nearly complete mapping of Event-B to the SAL input language. We do not consider it practical to attempt to cover the whole of the

Club				
Parameters	*States*	ProB	*Sal, old*	*SAL, new*
capacity = 1..2 set size = 5 MaxInt = 10	2408	9.5 sec	4 min 29 sec	0.3 sec
with symmetry	216	≈ 1.5 sec		
capacity = 1..2 set size = 6 MaxInt = 10	5857	39.4 sec		0.4 sec
with symmetry	245	≈ 2.5 sec		
capacity = 1..2 set size = 8 MaxInt = 10	31738	>10 min		0.7 sec
with symmetry	288	≈ 4 sec		
Scheduler				
Parameters	*States*	*ProB*	*Sal, old*	*SAL, new*
set size = 5	437	1.7 sec	55 sec	0.4 sec
with symmetry	21	0.25 sec		
set size = 7	5231	39.2 sec		0.7 sec
with symmetry	36	0.5 sec		

Table 2. The comparative results for the efficient set translation (Sal, new)

Event-B mathematical language. Instead, we intend for our tool to cooperate with the ProB model checker so that models that cannot be handled with SAL are automatically handled by ProB.

The result of developing the language mapping would be an automated translation of Event-B models into SAL. The next step is in providing a user with a meaningful feedback from the tool.

The summary of our translation approach is given in Table 3. It covers the main Event-B model elements such as events, invariant, variables, etc.

Since SAL requires variable types to be predefined and finite, all variables of an Event-B model must be automatically constrained to finite (and small) ranges. In Event-B models, definition of a variable type is a part of invariant. Therefore, constraints on variables can be obtained by analysing the invariant. In case of unbound type the model translator would use predefined ranges either specified by user or taken by default.

SAL supports enumerated types which can be used for encoding Event-B given sets. However, considering our general implementation of sets we see reasonable to map enumerated values into a range of integers at the translation level. The result of a SAL model checking would be traced back to the Event-B enumeration and given to a user as a feedback in terms of the initial model.

Along with solutions to translating Event-B models into SAL, we have identified a number of challenges:

- the use of cartesian products, relational composition and related operators often leads to a state explosion even in examples with modest model state space;
- it is apparent that some constructs of Event-B, such as closure, set comprehension and others, are very hard to translate into anything that would not preclude checking of interesting model properties;
- for some language constructs, it is easier to do partial translation. That is, we choose to assume that some properties hold without checking them to gain a performance benefit. This, for example, happens when accessing a function - there is no check of well-formedness of a function construct. Our intention is to benefit for the Rodin platform static checker and theorem prover to simplify translation by relying on the properties of a model already demonstrated by a static checker or a theorem prover.

General scheme	
Event-B	*SAL*
Model + context	Single module within a single SAL context
Events	Named guarded commands in TRANSITION block of a module
Event guards	Transition guards
Non-deterministic choice (ANY)	Variable becomes a parameter of a guarded command, the predicate becomes a part of a guard. The event is splitted into two transitions if necessary.
Invariants	Constraints on variables and sets go into type definition, remaining becomes a theorem
Contexts	
Carrier sets	Enumerated types or arrays
Constants	Constants
Named properties	Part of type definition, and guard of initialization transition
Types	
Basic types	Subranges of equivalent SAL types. Subranging depends on model being translated and model checking parameters.
Sets + operations	Arrays of reasonable size and operations on them
Total functions + operations	Total functions
Partial functions, injections, surjection + operations	Either definition in predicates or optimized using arrays

Table 3. Translation scheme

6. Related Work

Preliminary experience with translating Event-B to Alloy is reported in [22], but empirical results are not available yet. The authors also encountered the problem encoding complicated expressions of B in Alloy:

> "Expressions are the hardest part to encode. There is not only a myriad of complex expressions in Event-B but given that Alloy uses only flat relations, some Event-B expressions that introduce relations with nested sets generate many (and potentially large) Alloy expressions." [22]

Note that Daniel Jackson's dream was to work directly in Z, whose notation of expressions and predicates is very similar to B, but he abandoned that goal and developed the Alloy language

much more suitable for automated analysis.[3] Hence, it is no wonder that translating Z or B into Alloy is not trivial. The Z2SAL [10] project has similar goal as the translation to SAL shown here. Initial translations were not very efficient (see discussion in [20]), but more recent translations seem to perform better.

7. Conclusion and Future Work

We presented three approaches to model check B or Event-B specifications by translating them to other formalisms. A first naïve attempt to use low-level methods like BDDs soon turned out to be impractical and has no apparent advantage over using a more sophisticated tool.

SAL and Kodkod both are very promising. There is still room for improvements of the translation to support more expressions and to enhance the efficiency of the translated model. For some scenarios, using those translations gives us a much faster model check than PROB currently offers. On the other hand, for some tasks the constraint-solving approach of PROB is much more efficient (e.g., when working on large relations). As such, we believe there is considerable advantage in trying to combine these approaches (rather than one approach supplanting the other).

For SAL there is still the problem in translating more complicated Event-B data structures efficiently. Indeed, we have found that the SAL input language imposes a number of restrictions on what can be translated and how it is translated. For some objects, such as functions, the semantic gap between native Event-B formulae and SAL translation makes the interpretation of SAL output a bigger challenge than it could be. In addition, with a complex translation one has to worry about validity of the translation rules. As a way to overcome these limitations we are considering the possibility of translating a subset of Event-B mathematical language directly into Yices. The resulting tool could be used to improve the performance of PROB model checker and also to build a more capable disprover plugin [21].

References

[1] J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[2] J.-R. Abrial, M. Butler, and S. Hallerstede. An open extensible tool environment for Event-B. In *ICFEM06*, LNCS 4260, pages 588–605. Springer, 2006.

[3] Y. A. Ameur, F. Boniol, and V. Wiels, editors. *ISoLA 2007, Workshop On Leveraging Applications of Formal Methods, Verification and Validation, Poitiers-Futuroscope, France, December 12-14, 2007*, volume RNTI-SM-1 of *Revue des Nouvelles Technologies de l'Information*. Cépaduès-Éditions, 2007.

[4] J. Bendisposto, M. Leuschel, O. Ligot, and M. Samia. La validation de modèles event-b avec le plug-in prob pour rodin. *Technique et Science Informatiques*, 27(8):1065–1084, 2008.

[5] E. Börger, M. Butler, J. P. Bowen, and P. Boca, editors. *Abstract State Machines, B and Z, First International Conference, ABZ 2008, London, UK, September 16-18, 2008. Proceedings*, volume 5238 of *Lecture Notes in Computer Science*. Springer, 2008.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, Jun 1992.

[7] M. Butler and M. Leuschel. Combining CSP and B for specification and property verification. In *Proceedings of Formal Methods 2005*, LNCS 3582, pages 221–236, Newcastle upon Tyne, 2005. Springer-Verlag.

[8] S. Das and D. L. Dill. Successive approximation of abstract transition relations. In *Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science*, 2001. June 2001, Boston, USA.

[9] J. Derrick, S. North, and A. J. H. Simons. Z2sal - building a model checker for z. In Börger et al. [5], pages 280–293.

[10] J. Derrick, S. North, and T. Simons. Issues in implementing a model checker for Z. In Z. Liu and J. He, editors, *ICFEM*, LNCS 4260, pages 678–696. Springer, 2006.

[11] T. Hörne and J. A. van der Poll. Planning as model checking: the performance of prob vs nusmv. In R. Botha and C. Cilliers, editors, *SAICSIT Conf.*, volume 338 of *ACM International Conference Proceeding Series*, pages 114–123. ACM, 2008.

[12] D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11:256–290, 2002.

[13] M. Leuschel. The high road to formal validation. In Börger et al. [5], pages 4–23.

[14] M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[15] M. Leuschel, M. Butler, C. Spermann, and E. Turner. Symmetry reduction for B by permutation flooding. In *Proceedings B2007*, LNCS 4355, pages 79–93, Besancon, France, 2007. Springer-Verlag.

[16] M. Leuschel and M. J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.

[17] M. Leuschel, J. Falampin, F. Fritz, and D. Plagge. Automated property verification for large scale b models. submitted, 2009.

[18] M. Leuschel and M. Fontaine. Probing the depths of CSP-M: A new FDR-compliant validation tool. In *Proceedings ICFEM 2008*, LNCS, pages 278–297. Springer-Verlag, 2008.

[19] M. Leuschel and T. Massart. Efficient approximate verification of B via symmetry markers. In *Proceedings International Symmetry Conference*, pages 71–85, Edinburgh, UK, January 2007.

[20] M. Leuschel and D. Plagge. Seven at a stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. In Ameur et al. [3], pages 73–84.

[21] O. Ligot, J. Bendisposto, and M. Leuschel. Debugging Event-B Models using the ProB Disprover Plug-in. *Proceedings AFADL'07*, Juni 2007.

[22] P. J. Matos and J. Marques-Silva. Model checking event-b by encoding into alloy. In Börger et al. [5], page 346.

[23] D. Plagge and M. Leuschel. Validating Z Specifications using the ProB Animator and Model Checker. In J. Davies and J. Gibbons, editors, *Proceedings IFM 2007*, LNCS 4591, pages 480–500. Springer-Verlag, 2007.

[24] The SAL website. `http://sal.csl.sri.com`.

[25] G. Smith and L. Wildman. Model checking Z specifications using SAL. In *ZB*, pages 85–103, 2005.

[26] E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *TACAS*, LNCS 4424, pages 632–647. Springer, 2007.

[27] E. Turner, M. Leuschel, C. Spermann, and M. Butler. Symmetry reduced model checking for B. In *Proceedings Symposium TASE 2007*, pages 25–34, Shanghai, China, June 2007. IEEE.

[28] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM Press.

[3] Invited talk "Recent Advances in Alloy" at iFM 2007 in Oxford.

Automatic Synthesis of an Efficient Algorithm for the Similarity of Strings Problem

Luca Chiarabini
LMU Mathematisches Institut, Theresienstrasse 39, D-80333 München, Germany
chiarabi@mathematik.uni-muenchen.de

ABSTRACT

In this paper we propose a method for the automatic synthesis an efficient program for the *similarity of strings* problem. The synthesis is performed by the proof assistant MINLOG, a proof assistant for the machine-extraction of realistic programs from proofs. The originality of the proposed method rely on the fact that the extracted algorithm is in dynamic programming style. The method we propose can not be applied automatically to an arbitrary proof; it can be seen more as a general schema (that has to be instantiated case by case) to follow in order extract dynamic programs from proofs.

1. INTRODUCTION

A widely studied problem in bioinformatic is to find the *distance* between two given sequences of symbols (over an alphabet Σ) The two main technique developed in this area to solve this problem turned out to be the *edit distance* and the *similarity* of strings [1].

Edit distance focus on the transformation of the first list into the second one using a restricted set of operations (insertion I, deletion D, matching M, and replacement R) Given two lists we define the *edit distance problem* the task of finding the minimum number of insertions, deletions and substitutions operations to transform the first list to the second one. Once the right set of basic operation is found, this is stored in a string called *edit transcript* (build on the alphabet I,D,M, and R) that will constitute the output of the problem (Figure 1, line 1).

The other way to measure the distance of lists is the so called *similarity* method. The idea is based on the concept of *string alignment*. Given two strings l_1 and l_2, *an* alignment of l_1 and l_2 is obtained inserting a new symbol "_" (named *space*) (that does not belong to Σ) into the strings l_1 and l_2 and then placing the two strings one above the other, so that every character or space in either list is opposite a unique character or space in the other list, and no space is opposite to another space (Figure 1, lines 2,3). We indicate by (δ_1, δ_2) a general alignment the lists l_1 and l_2. Here δ_1 and δ_2 are strings over $\Sigma \cup \{_\}$. Afterwards the similarity between l_1 and l_2 is defined as the greatest $\mathcal{E}((\delta_1, \delta_2))$ with $\mathcal{E}$ function with values in $\mathbf{N}$ that associate a score to each alignment (δ_1, δ_2).

$$\begin{array}{lccccccccc} \mathbf{1}: & \mathtt{R} & \mathtt{I} & \mathtt{M} & \mathtt{D} & \mathtt{M} & \mathtt{D} & \mathtt{M} & \mathtt{M} & \mathtt{I} \\ \mathbf{2}: & v & _ & i & n & t & n & e & r & _ \\ \mathbf{3}: & w & r & i & _ & t & _ & e & r & s \end{array} \quad (1)$$

Figure 1: Alignment (lines 2, 3) and edit-transcript (line 1) of the strings *wintner* and *writers*. It is possible to note how the two methods are equivalent: a mismatch in the alignment correspond to a replacement in the edit transcript, a space in the alignment contained in the first string correspond to the insertion of the opposite character in first string, and a space in the alignment contained in the second string correspond to a deletion of the opposite character in the first string.

In computational biology the similarity of l_1 and l_2 is efficiently solved using dynamic programming; in fact the problem can be solved storing in a matrix M, of dimension $|l_1| \times |l_2|$, the values of the similarities between all the prefixes of length $i \leq |l_1|$ and $j \leq |l_2|$ of l_1 and l_2. This could be seen as a sort of generalization of the Fibonacci problem to 2-dimensions.

In this work we will formalize the *similarity* problem in the proof assistant MINLOG (www.minlog-system.de). MINLOG is intended to reason about computable functions of finite type using minimal logic. A major aim of the MINLOG project is the development of practically useful tools for the machine-extraction of realistic programs from proofs. We will extract, from the proof of the existence of an alignment with highest score between two given strings the naive exponential program to compute the similarity of strings. Afterwards, we will propose a method to transform the given proof into another from which it will be possible to extract a more efficient program, in dynamic programming style.

We propose a method that we name *list as memory*. The idea consist in evaluating a sufficient amount of data in advance so that the extracted algorithm gets to reuse it instead of recomputing it each time it is needed. This is done introducing in the proof a list of *ad-hoc* axioms. The method we propose can not be applied automatically to an arbitrary

©ACM, (2009). This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in PUBLICATION, {VOL#, ISS#,(DATE)} http://doi.acm.org/10.1145/{nnnnnn.nnnnnn }
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

proof; it can be seen more as a general schema (that has to be instantiated case by case) to follow in order extract dynamic programs from proofs.

The paper is organized as follow: in section 2 we revisit the basic logical notions regarding the program extraction from constructive proofs, in section 3 we formalize the proof of the existence of an alignment with highest score between lists and we extract a program from the proof. The designed solution enumerate all the alignments in order to find the right one, and this generate an exponential running time algorithm. In section 4 we present a proof transformation to apply to the proof presented in section 3, in order to extract an algorithm in dynamic programming style. In section 5, we make some final considerations over the presented method and future works.

2. MODIFIED REALIZABILITY FOR FIRST ORDER MINIMAL LOGIC

2.1 Gödel's T

Types are built from base types $\mathbf{N}$ (Naturals) , $\mathbf{L}(\rho)$ (lists with elements of type ρ) and $\mathbf{B}$ (booleans) by function ($\rightarrow$) and pair ($\times$) formation. The *Terms* of Gödel's T [2] are simply typed λ-calculus terms with pairs, projections (π_i) and constants (constructors and recursive operators for the basic types)

$$\begin{array}{rcl} \textit{Types } \rho,\sigma & ::= & \mathbf{N}\,|\,\mathbf{B}\,|\,\mathbf{L}(\rho)\,|\,\rho\rightarrow\sigma\,|\,\rho\times\sigma \\ \textit{Const } c & ::= & 0^{\mathbf{N}}\,|\,\mathsf{Succ}^{\mathbf{N}\rightarrow\mathbf{N}}\,|\,\mathsf{tt}^{\mathbf{B}}\,|\,\mathsf{ff}^{\mathbf{B}}\,|\,(:)^{\mathbf{L}(\rho)}\,|\,::^{\rho\rightarrow\mathbf{L}(\rho)\rightarrow\mathbf{L}(\rho)}\,| \\ & & \mathcal{R}^{\sigma}_{\mathbf{N}}|\,\mathcal{R}^{\sigma}_{\mathbf{L}(\rho)}\,|\mathcal{R}^{\sigma}_{\mathbf{B}} \\ \textit{Terms } r,s,t & ::= & c\,|x^{\rho}|(\lambda x^{\rho}r^{\sigma})^{\rho\rightarrow\sigma}|(r^{\rho\rightarrow\sigma}s^{\rho})^{\sigma}|(\pi_0 t^{\rho\times\sigma})^{\rho}| \\ & & (\pi_1 t^{\rho\times\sigma})^{\sigma}\,|\,(r^{\rho},s^{\sigma})^{\rho\times\sigma} \end{array}$$

The expression (:) represents the empty list, and (a_0 :: $\ldots$ :: a_n :) a list with $n+1$ elements. We equip this calculus with the usual conversion rules for the recursive operators, applications and projections (Figure 2).

In the MINLOG proof assistant, extracted programs are presented in a textual style, that we briefly describe now along with the correspondence with the above mathematical notations: in programs produced by MINLOG, tt and ff are typeset `#tt` and `#ff` respectively; $\rho\times\sigma$ as `(rho@@sigma)`, $\mathbf{L}(\rho)$ as `(list rho)`, $\lambda x.t$ is written as `([x]t)`, $(\mathcal{R}^{\sigma}_{\mathbf{N}/\mathbf{B}/\mathbf{L}(\rho)}\,b\,s)$ as `(Rec (nat/bool/list rho => sigma) b s)` and $(\pi_{0/1}e)$ as `(left/right e)`. Finally the term $(\mathcal{R}^{\sigma}_{\mathbf{B}}\,r\,s)t$ is printed as `(if t r s)`.

2.2 Heyting Arithmetic

We define Heyting Arithmetic HA$^{\omega}$ for our language based on Gödel's T, which is finitely typed. We define *negation* $\neg\varphi$ by $\varphi\rightarrow\bot$.

Formulas: Atomic formulas ($\mathrm{P}\vec{t}^{\vec{\rho}}$) (P a predicate symbol, $\vec{t}$, $\vec{\rho}$ lists of terms and types), $\varphi\rightarrow\psi$, $\forall x^{\rho}\varphi$, $\exists x^{\rho}\varphi$, $\varphi\wedge\psi$.

Derivations: By the Curry-Howard correspondence it is convenient to write derivations as terms: we define λ-terms M^{φ} for natural deduction proofs of formulas φ together with the set OA(M) of *o*pen *a*ssumptions in M.

$$\begin{array}{ll} (ass) & u^{\varphi},\ \mathrm{OA}(u)=\{u\} \\ (\wedge^+) & (\langle M^{\varphi},N^{\psi}\rangle^{\varphi\wedge\psi}),\ \mathrm{OA}(\langle M,N\rangle)=\mathrm{OA}(M)\cup\mathrm{OA}(N) \\ (\wedge_0^-) & (M^{\varphi\wedge\psi}0)^{\varphi}\ ,\ \mathrm{OA}(M0)=\mathrm{OA}(M) \\ (\wedge_1^-) & (N^{\varphi\wedge\psi}1)^{\psi}\ ,\ \mathrm{OA}(N1)=\mathrm{OA}(N) \\ (\rightarrow^+) & (\lambda u^{\varphi}M^{\psi})^{\varphi\rightarrow\psi}\ ,\ \mathrm{OA}(\lambda uM)=\mathrm{OA}(M)\backslash\{u\} \\ (\rightarrow^-) & (M^{\varphi\rightarrow\psi}N^{\varphi})^{\psi},\ \mathrm{OA}(MN)=\mathrm{OA}(M)\cup\mathrm{OA}(N) \\ (\forall^+) & (\lambda x^{\rho}M^{\varphi})^{\forall x^{\rho}\varphi},\ \mathrm{OA}(\lambda xM)=\mathrm{OA}(M) \\ & \text{provided } x^{\rho}\notin\mathrm{FV}(\varphi),\text{ for any } u^{\varphi}\in\mathrm{OA}(M) \\ (\forall^-) & (M^{\forall x^{\rho}\varphi}t^{\rho})^{\varphi},\ \mathrm{OA}(Mt)=\mathrm{OA}(M) \end{array}$$

Usually we will omit type and formula indices in derivations if they are uniquely determined by the context or if they are not relevant.

We use $\exists$ and $\vee$ in our logic, if we allow appropriate axioms as constant derivation terms:

$$\begin{array}{rcl} \exists^+_{x^{\rho},\varphi} & : & \forall x^{\rho}(\varphi\rightarrow\exists x^{\rho}\varphi) \\ \exists^-_{x^{\rho},\varphi,\psi} & : & \exists x^{\rho}\varphi\rightarrow(\forall x^{\rho}\varphi\rightarrow\psi)\rightarrow\psi \text{ with } \notin FV(\psi) \end{array}$$

We use the following axioms to perform proofs by induction over naturals ($\mathbf{N}$), booleans ($\mathbf{B}$) and lists of elements of type ρ ($\mathbf{L}(\rho)$):

$$\begin{array}{lcl} \mathsf{Ind}_{n,A} & : & A[n\mapsto 0]\rightarrow(\forall n.A\rightarrow A[n\mapsto \mathrm{S}n])\rightarrow\forall n^{\mathbf{N}}A, \\ \mathsf{Ind}_{p,A} & : & A[p\mapsto \mathsf{tt}]\rightarrow A[p\mapsto \mathsf{ff}]\rightarrow\forall p^{\mathbf{B}}A, \\ \mathsf{Ind}_{l,A} & : & A[l\mapsto \mathrm{nil}]\rightarrow(\forall x,l.A\rightarrow A[l\mapsto \mathrm{cons}(x,l)])\rightarrow\forall l^{\mathbf{L}(\alpha)}A \end{array}$$

Finally we use the constant derivation term (IF_{φ}),

$$\mathsf{IF}_{\varphi}:\forall p^{\mathbf{B}}(p\rightarrow\varphi)\rightarrow((p\rightarrow\bot)\rightarrow\varphi)\rightarrow\varphi$$

to perform *case distinction* on boolean terms w.r.t. a goal formula φ.

2.3 Short Excursus in Program Extraction from Proofs

Clearly proper existence proofs have computational content. A well-known and natural way to define this concept is the notion of realizability, which can be seen as an incarnation of the Brouwer-Heyting-Kolmogorov interpretation of proofs.

2.3.1 Type of a Formula

We indicate by $\tau(\varphi)$ as the type of the term (or "program") to be extracted from a proof of φ. More precisely, to every formula φ is possible to assign an object $\tau(\varphi)$ (a type or the "nulltype" symbol ε). In case $\tau(\varphi)=\varepsilon$ proofs of φ have no computational content; such formulas φ are called *Harrop formulas*.

$$\tau(P(\vec{x})) = \begin{cases} \alpha_P & \text{if } P \text{ is a predicate variable with assigned } \alpha_P \\ \varepsilon & \text{Otherwise} \end{cases}$$

$$\tau(\exists x^{\rho}\varphi) = \begin{cases} \rho & \text{if } \tau(\varphi)=\varepsilon \\ \rho\times\tau(\varphi) & \text{Otherwise} \end{cases}$$

$$\tau(\forall x^{\rho}\varphi) = \begin{cases} \varepsilon & \text{if } \tau(\varphi)=\varepsilon \\ \rho\rightarrow\tau(\varphi) & \text{Otherwise} \end{cases}$$

$$\tau(\varphi\wedge\psi) = \begin{cases} \tau(\varphi) & \text{if } \tau(\psi)=\varepsilon \\ \tau(\psi) & \text{if } \tau(\varphi)=\varepsilon \\ \tau(\varphi)\times\tau(\psi) & \text{Otherwise} \end{cases}$$

$$\tau(\varphi\rightarrow\psi) = \begin{cases} \tau(\psi) & \text{if } \tau(\varphi)=\varepsilon \\ \varepsilon & \text{if } \tau(\psi)=\varepsilon \\ \tau(\varphi)\rightarrow\tau(\psi) & \text{Otherwise} \end{cases}$$

$$\mathcal{R}_{\mathbf{N}}^{\sigma} : \sigma \to (\mathbf{N} \to \sigma \to \sigma) \to \mathbf{N} \to \sigma$$
$$(\mathcal{R}_{\mathbf{N}}^{\sigma}\, b\, f)\, 0 \longmapsto b$$
$$(\mathcal{R}_{\mathbf{N}}^{\sigma}\, b\, f)\, (n+1) \longmapsto f\, n\, ((\mathcal{R}_{\mathbf{N}}^{\sigma}\, b\, f)\, n)$$

$$\mathcal{R}_{\mathbf{B}}^{\sigma} : \sigma \to \sigma \to \mathbf{B} \to \sigma$$
$$(\mathcal{R}_{\mathbf{B}}^{\sigma}\, r\, s)\mathsf{tt} \longmapsto r$$
$$(\mathcal{R}_{\mathbf{B}}^{\sigma}\, r\, s)\mathsf{ff} \longmapsto s$$

$$\mathcal{R}_{\mathbf{L}(\rho)}^{\sigma} : \sigma \to (\rho \to \mathbf{L}(\rho) \to \sigma \to \sigma) \to \mathbf{L}(\rho) \to \sigma$$
$$(\mathcal{R}_{\mathbf{L}(\rho)}^{\sigma}\, b\, f)\, [] \longmapsto b$$
$$(\mathcal{R}_{\mathbf{L}(\rho)}^{\sigma}\, b\, f)\, (a :: l) \longmapsto f\, l\, ((\mathcal{R}_{\mathbf{L}(\rho)}^{\sigma}\, b\, f)\, l)$$

$$\pi_0(r,s) \longrightarrow r$$
$$\pi_1(r,s) \longrightarrow s$$
$$(\lambda x.r)s \longrightarrow r[x := s]$$

Figure 2: Conversion rules for typed Gödel T

2.3.2 *Extraction Map*

From every derivation M of a computationally meaningful formula φ (that is, $\tau(\varphi) \neq \varepsilon$) is possible to define its *extracted program* $[\![M]\!]$ of type $\tau(\varphi)$[3]. If $\tau(\varphi) = \varepsilon$ then $[\![M]\!] = \varepsilon$.

$$\begin{aligned}
[\![u^{\varphi}]\!] &= x_u^{\varphi} \quad (x_u^{\varphi} \text{ uniquely associated with } \varphi) \\
[\![\lambda u^{\varphi} M]\!] &= \begin{cases} [\![M]\!] & \text{if } \tau(\varphi) = \varepsilon \\ \lambda x_u^{\tau(\varphi)} [\![M]\!] & \text{Otherwise} \end{cases} \\
[\![M^{\varphi \to \psi} N^{\psi}]\!] &= \begin{cases} [\![M]\!] & \text{if } \tau(\varphi) = \varepsilon \\ [\![M]\!][\![N]\!] & \text{Otherwise} \end{cases} \\
[\![\langle M^{\varphi}, N^{\psi} \rangle]\!] &= \begin{cases} [\![N]\!] & \text{if } \tau(\varphi) = \varepsilon \\ [\![M]\!] & \text{if } \tau(\psi) = \varepsilon \\ \langle [\![M]\!], [\![N]\!] \rangle & \text{Otherwise} \end{cases} \\
[\![M^{\varphi \wedge \psi} i]\!] &= \begin{cases} [\![M]\!] & \text{if } \tau(\varphi) = \varepsilon \text{ or } \tau(\psi) = \varepsilon \\ \pi_i [\![M]\!] & \text{if Otherwise} \end{cases} \\
[\![(\lambda x^{\rho} M)^{\forall x \varphi}]\!] &= \lambda x^{\rho} [\![M]\!] \\
[\![M^{\forall x \varphi} t]\!] &= [\![M]\!] t
\end{aligned}$$

Content of the proof constants:

$$\begin{aligned}
[\![\exists^-_{x^{\rho}, \varphi, \psi}]\!] &= \begin{cases} \lambda x^{\rho} f^{\rho \to \tau(\psi)}.fx & \text{If } \tau(\varphi) = \varepsilon \\ \lambda x^{\rho \times \tau(\varphi)} f^{\rho \to \tau(\varphi) \to \tau(\psi)}.f(\pi_0 x)(\pi_1 x) & \text{Otherwise} \end{cases} \\
[\![\exists^+_{x^{\rho}, \varphi}]\!] &= \begin{cases} \lambda x^{\rho} x & \text{If } \tau(\varphi) = \varepsilon \\ \lambda x^{\rho} y^{\tau(\varphi)}.\langle x, y \rangle & \text{Otherwise} \end{cases} \\
[\![\mathrm{Ind}_{n, \varphi(n)}]\!] &= \mathcal{R}_{\mathbf{N}}^{\sigma} \\
[\![\mathrm{Ind}_{l, \varphi(l)}]\!] &= \mathcal{R}_{\mathbf{L}(\rho)}^{\sigma} \\
[\![\mathrm{Ind}_{t, \varphi(t)}]\!] &= \mathcal{R}_{\mathbf{B}}^{\sigma}
\end{aligned}$$

2.3.3 *Realize a formula*

Correctness of the extracted programs is guaranteed by the notion of *modified realizability*. Intuitively, if t is the extracted program from the derivation M of the formula φ equal to $\forall x \exists y.P(x,y)$ then for each x the formula $P(x, t(x))$ is provable correct (*Soundness*) i.e. t *(modified) realize* φ (written $(t\, \boldsymbol{mr}\, \varphi)$)

$$\begin{aligned}
r\, \mathbf{mr}\, P(\vec{t}) &= P(\vec{t}) \\
r\, \mathbf{mr}\, (\exists x.\varphi) &= \begin{cases} \varepsilon\, \mathbf{mr} \varphi[x/r] & \text{if } \tau(\varphi) = \varepsilon \\ \pi_1 r\, \mathbf{mr}\, \varphi[x/\pi_0 r] & \textit{Otherwise} \end{cases} \\
r\, \mathbf{mr}\, (\forall x.\varphi) &= \begin{cases} \forall x.\varepsilon\, \mathbf{mr}\, \varphi & \text{if } \tau(\varphi) = \varepsilon \\ \forall x.rx\, \mathbf{mr}\, \varphi & \textit{Otherwise} \end{cases} \\
r\, \mathbf{mr}\, (\exists^{\mathsf{nc}} x.\varphi) &= \begin{cases} \exists^{\mathsf{nc}} x.\varepsilon\, \mathbf{mr} \varphi & \text{if } \tau(\varphi) = \varepsilon \\ \exists^{\mathsf{nc}} x.r\, \mathbf{mr}\, \varphi & \textit{Otherwise} \end{cases} \\
r\, \mathbf{mr}\, (\forall x^{\mathsf{nc}}.\varphi) &= \begin{cases} \forall^{\mathsf{nc}} x.\varepsilon\, \mathbf{mr}\, \varphi & \text{if } \tau(\varphi) = \varepsilon \\ \forall^{\mathsf{nc}} x.r\, \mathbf{mr}\, \varphi & \textit{Otherwise} \end{cases} \\
r\, \mathbf{mr}\, (\varphi \to \psi) &= \begin{cases} \varepsilon\, \mathbf{mr}\, \varphi \to r\, \mathbf{mr}\, \psi & \text{if } \tau(\varphi) = \varepsilon \\ \forall x.x\, \mathbf{mr}\, \varphi \to \varepsilon\, \mathbf{mr}\, \psi & \text{if } \tau(\varphi) \neq \varepsilon = \tau(\psi) \\ \forall x.x\, \mathbf{mr}\, \varphi \to rx\, \mathbf{mr}\, \psi & \textit{Otherwise} \end{cases} \\
r\, \mathbf{mr}\, (\varphi \wedge \psi) &= \begin{cases} \varepsilon\, \mathbf{mr}\, \varphi \wedge r\, \mathbf{mr}\, \psi & \text{if } \tau(\varphi) = \varepsilon \\ r\, \mathbf{mr}\, \varphi \to \varepsilon\, \mathbf{mr}\, \psi & \text{if } \tau(\psi) = \varepsilon \\ \pi_0 r\, \mathbf{mr}\, \varphi \to \pi_1 r\, \mathbf{mr}\, \psi & \textit{Otherwise} \end{cases}
\end{aligned}$$

THEOREM 2.1 (SOUNDNESS). *Let M be a derivation of a formula φ from assumptions $u_i : \varphi_i$. Then we can find a derivation of the formula ($[\![M]\!]$ $\boldsymbol{mr}$ φ) from assumptions $\bar{u}_i : x_{u_i}$ $\boldsymbol{mr}$ φ_i.*

PROOF. Induction on M[4]. □

3. FORMALIZATION OF THE SIMILARITY PROBLEM

Let l_1 and l_2 be two lists built on the alphabet Σ, with Σ equal to $\mathbf{N}_{>0}$ (the set of naturals strictly higher than zero), $0 \notin \Sigma$ be the *space* character and $\alpha : \mathbf{N} \times \mathbf{N} \to \mathbf{Z}$ be a *scoring function*.

Given two lists l_1 and l_2 over Σ, in figure 3 we give an inductive definition of the family of sets $\mathcal{A}_{i,j}^{l_1,l_2}$, the *alignments* of the first $i \leq |l_1|$ characters of l_1 and $j \leq |l_2|$ characters of l_2.

In figure 3 and in the rest of the paper we make use of the following conventions: n, m, i and j ranges over $\mathbf{N}$, $|l|$ is the length of l, $l[i]$ is the $i+1$-th character of l, $\mathsf{head}(a :: l) = a$, $\mathsf{tail}(a :: l) = l$, $pre_n(l)$ is a partial operator that return the first n elements of l, 0^n is the list composed by a sequence of n zeros, $l \cdot g$ the operation of appending the list g to l and $(a_1, \ldots, a_n)$ is the list composed by $a_i \in \mathbf{N}$.

We associate a *score* to each alignment by the *evaluator*

$$(A_0)\ \frac{}{((:),(:)) \in \mathcal{A}^{l_1,l_2}_{0,0}} \qquad (A_1)\ \frac{}{(0^{j+1}, pre_{j+1}(l_2)) \in \mathcal{A}^{l_1,l_2}_{0,j+1}} \qquad (A_2)\ \frac{}{(pre_{i+1}(l_1), 0^{i+1}) \in \mathcal{A}^{l1,l2}_{i+1,0}}$$

$$(A_3)\ \frac{(\delta_1,\delta_2) \in \mathcal{A}^{l_1,l_2}_{i+1,j}}{(\delta_1 \cdot (0:), \delta_2 \cdot l_2[j]) \in \mathcal{A}^{l_1,l_2}_{i+1,j+1}} \qquad (A_4)\ \frac{(\delta_1,\delta_2) \in \mathcal{A}^{l_1,l_2}_{i,j+1}}{(\delta_1 \cdot l_1[i], \delta_2 \cdot (0:)) \in \mathcal{A}^{l_1,l_2}_{i+1,j+1}} \qquad (A_5)\ \frac{(\delta_1,\delta_2) \in \mathcal{A}^{l_1,l_2}_{i,j}}{(\delta_1 \cdot l_1[i], \delta_2 \cdot l_2[j]) \in \mathcal{A}^{l_1,l_2}_{i+1,j+1}}$$

Figure 3: Induction definition of the alignments $\mathcal{A}^{l_1,l_2}_{|l_1|,|l_2|}$

function $\mathcal{E} : \mathcal{A}^{l_1,l_2}_{i,j} \to (\mathbf{N} \times \mathbf{N} \to \mathbf{N}) \to \mathbf{N}$ defined on the inductive structure of $\mathcal{A}^{l_1,l_2}_{i,j}$ (Figure 4). The function $\mathcal{E}$ takes in input an alignment, a scoring function and return the score of the input alignment. Our goal is to find the alignment in $\mathcal{A}^{l_1,l_2}_{|l_1|,|l_2|}$ with highest score (this score will be the *similarity* between l_1 and l_2) with respect to a given scoring function α.

Remark Many problems can be modeled as special case of similarity by choosing an appropriate scoring function α. Let consider (below) the definition of the *longest common subsequence problem.*

Definition 3.1. *A subsequence of a string l is specified by a list of indices $i_1 < i_2 < \ldots < i_k$ for some $k \leq |l|$. The subsequence specified by this list is the string $l[i_1]l[i_2]\ldots l[i_k]$*

Definition 3.2 (Longest Common SubsequenceProblem). *Given two strings l_1 and l_2 a common subsequence of l_1 and l_2 is a sequence that appear both in l_1 and l_2. The Longest Common Subsequence Problem consist in finding the longest common subsequence between l_1 and l_2*

For example, 145 is a common subsequence of 114666725 and 1124375 but 11475 is the longest common ones. The solution of the longest common subsequence problem can be obtained from the solution of the similarity of lists problem by choosing a scoring function α that scores a "1" for each *match* and "0" for each *mismatch* or presence of a 0 (the result will depend by the implemented strategy to solve the problem since there could be more alignments with the same highest score).

Now we show formally that given a couple of lists l_1, l_2 over Σ there exists always an alignment in $\mathcal{A}^{l_1,l_2}_{|l_1|,|l_2|}$ of maximum score with respect to α.

Theorem 3.1.
$\forall l_1, l_2 \exists \delta_1, \delta_2((\delta_1,\delta_2) \in \mathcal{A}^{l_1,l_2}_{|l_1|,|l_2|}) \wedge$
$\forall \delta_1', \delta_2'((\delta_1',\delta_2') \in \mathcal{A}^{l_1,l_2}_{|l_1|,|l_2|} \to \mathcal{E}(\delta_1',\delta_2') \leq \mathcal{E}(\delta_1,\delta_2))$

Proof. We assume l_1 and l_2. In order to prove the thesis we prove the following statement:

$\forall n, m \exists \delta_1, \delta_2((\delta_1,\delta_2) \in \mathcal{A}^{l_1,l_2}_{n,m}) \wedge$
$\forall \delta_1', \delta_2'((\delta_1',\delta_2') \in \mathcal{A}^{l_1,l_2}_{n,m}) \to \mathcal{E}(\delta_1',\delta_2') \leq \mathcal{E}(\delta_1,\delta_2)$

Obviously we obtain the thesis instantiating this assertion on $|l_1|$ and $|l_2|$. From now on we will write $Q(\delta_1,\delta_2,n,m)$ for

$$((\delta_1,\delta_2) \in \mathcal{A}^{l_1,l_2}_{n,m}) \wedge \forall \delta_1', \delta_2'((\delta_1',\delta_2') \in \mathcal{A}^{l_1,l_2}_{n,m}) \to \mathcal{E}(\delta_1',\delta_2') \leq \mathcal{E}(\delta_1,\delta_2)$$

We go by induction on n and m.
Base Case$[n = 0]$ We prove

$$\forall m \exists \delta_1, \delta_2 Q(\delta_1,\delta_2,0,m)$$

by case distinction over m:

Base Case$[n = 0,\ m = 0]$: $Q((:),(:),0,0)$ by rule (A_0).
Induction Step$[n = 0,\ m + 1]$
We have $Q(0^{m+1}, pre_{m+1}(l_2), 0, m+1)$ by rule (A_1).

Induction Step$[n + 1]$ We now assume

$$\forall m \exists \delta_1, \delta_2 Q(\delta_1,\delta_2,n,m) \tag{2}$$

and we must show

$$\forall m \exists \delta_1, \delta_2 Q(\delta_1,\delta_2,n+1,m)$$

By induction over m:

Base Case$[n + 1,\ m = 0]$
$Q(pre_{n+1}(l_1), 0^{n+1}, n+1, 0)$ by (A_2)

Induction Step$[n + 1,\ m + 1]$: Assume

$$\exists \delta_1, \delta_2 Q(\delta_1,\delta_2,n+1,m) \tag{3}$$

we have to prove

$$\exists \delta_1, \delta_2 Q(\delta_1,\delta_2,n+1,m+1)$$

By (3) there exists δ_1', δ_2' such that $(\delta_1',\delta_2') \in \mathcal{A}^{l_1,l_2}_{n+1,m}$ and such that for every $(\delta_1,\delta_2) \in \mathcal{A}^{l_1,l_2}_{n+1,m}$

$$\mathcal{E}(\delta_1,\delta_2) \leq \mathcal{E}(\delta_1',\delta_2') \tag{4}$$

Instantiating (2) on $m + 1$ there exists δ_1'', δ_2'' such that $(\delta_1'',\delta_2'') \in \mathcal{A}^{l_1,l_2}_{n,m+1}$ and for every $(\delta_1,\delta_2) \in \mathcal{A}^{l_1,l_2}_{n,m+1}$

$$\mathcal{E}(\delta_1,\delta_2) \leq \mathcal{E}(\delta_1'',\delta_2'') \tag{5}$$

Instantiating (2) on m there exists δ_1''', δ_2''' such that $(\delta_1'',\delta_2'') \in \mathcal{A}^{l_1,l_2}_{n,m}$ and for every $(\delta_1,\delta_2) \in \mathcal{A}^{l_1,l_2}_{n,m}$

$$\mathcal{E}(\delta_1,\delta_2) \leq \mathcal{E}(\delta_1''',\delta_2''') \tag{6}$$

Now we have to dispatch over the following cases:

ip_1. $\mathcal{E}(\delta_1'' \cdot l_1[n+1], \delta_2'' \cdot (0:)) \leq \mathcal{E}(\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1])$:

$$(E_0)\ \frac{}{\mathcal{E}[((:),(:))]_\alpha = 0} \qquad (E_1)\ \frac{}{\mathcal{E}[(0^j, pre_j(l_2))]_\alpha = \sum_{k=1}^{j} \alpha(0, l_2[k])} \qquad (E_2)\ \frac{}{\mathcal{E}[(pre_i(l_1), 0^i)]_\alpha = \sum_{k=1}^{i} \alpha(l_1[k], 0)}$$

$$(E_3)\ \frac{\mathcal{E}[(\delta_1,\delta_2)] = n}{\mathcal{E}[(\delta_1 \cdot (0:), \delta_2 \cdot l_2[j])]_\alpha = n + \alpha(0, l_2[j])} \quad (E_4)\ \frac{\mathcal{E}[(\delta_1,\delta_2)] = n}{\mathcal{E}[(\delta_1 \cdot l_1[i], \delta_2 \cdot (0:))]_\alpha = n + \alpha(l_1[i], 0)} \quad (E_5)\ \frac{\mathcal{E}[(\delta_1,\delta_2)] = n}{\mathcal{E}[(\delta_1 \cdot l_1[i], \delta_2 \cdot l_2[j])]_\alpha = n + \alpha(l_1[i], l_2[j])}$$

Figure 4: Induction definition of the *evaluator* function $\mathcal{E}$

Then, only 2 cases are possible:

$\mathsf{ip}_{1.1}$. $\mathcal{E}(\delta_1''' \cdot l_1[n+1], \delta_2''' \cdot l_2[m+1]) \leq \mathcal{E}(\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1])$: We claim $Q(\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1], n+1, m+1)$. This is proved dispatching over (δ_1, δ_2) in $\mathcal{A}^{l_1,l_2}_{n+1,m+1}$. In fact for every $(\delta_1, \delta_2) \in \mathcal{A}^{l_1,l_2}_{n+1,m+1}$ only three cases are possible

$\mathsf{ip}_{1.1.1}$.$(\delta_1, \delta_2) = (\delta_1^* \cdot (0:), \delta_2^* \cdot l_2[m+1])$:

$$\begin{aligned} \mathcal{E}(\delta_1, \delta_2) &= \mathcal{E}(\delta_1^* \cdot (0:), \delta_2^* \cdot l_2[m+1]) \\ &= \mathcal{E}(\delta_1^*, \delta_2^*) + \alpha((0:), l_2[m+1]) \quad \text{by}(E_3) \\ &\leq \mathcal{E}(\delta_1', \delta_2') + \alpha((0:), l_2[m+1]) \quad \text{by (4)} \\ &= \mathcal{E}(\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1]) \quad \text{by}(E_3) \end{aligned}$$

$\mathsf{ip}_{1.1.2}$. $(\delta_1, \delta_2) = (\delta_1^* \cdot l_1[n+1]), \delta_2^* \cdot (0:))$:

$$\begin{aligned} \mathcal{E}[(\delta_1, \delta_2)] &= \mathcal{E}(\delta_1^* \cdot l_1[n+1], \delta_2^* \cdot (0:)) \\ &= \mathcal{E}(\delta_1^*, \delta_2^*) + \alpha(l_1[n+1], (0:)) \quad \text{by}(E_4) \\ &\leq \mathcal{E}(\delta_1'', \delta_2'') + \alpha(l_1[n+1], (0:)) \quad \text{by (5)} \\ &= \mathcal{E}(\delta_1'' \cdot l_1[n+1], \delta_2'' \cdot (0:)) \\ &\leq \mathcal{E}(\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1]) \quad \text{by}\ (\mathsf{ip}_1) \end{aligned}$$

$\mathsf{ip}_{1.1.3}$. $(\delta_1, \delta_2) = (\delta_1^* \cdot l_1[n+1]), \delta_2^* \cdot l_2[m+1])$:

$$\begin{aligned} \mathcal{E}[(\delta_1, \delta_2)] &= \mathcal{E}(\delta_1^* \cdot l_1[n+1], \delta_2^* \cdot l_2[m+1]) \\ &= \mathcal{E}(\delta_1^*, \delta_2^*) + \alpha(l_1[n+1], l_2[m+1]) \quad \text{by}(E_5) \\ &\leq \mathcal{E}(\delta_1''', \delta_2''') + \alpha(l_1[n+1], l_2[m+1]) \quad \text{by (6)} \\ &= \mathcal{E}(\delta_1''' \cdot l_1[n+1], \delta_2''' \cdot l_2[m+1]) \\ &\leq \mathcal{E}(\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1]) \quad \text{by}\ (\mathsf{ip}_{1.1}) \end{aligned}$$

$\mathsf{ip}_{1.2}$. $\mathcal{E}(\delta_1''' \cdot l_1[n+1], \delta_2''' \cdot l_2[m+1]) \not\leq \mathcal{E}(\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1])$: We claim $Q(\delta_1''' \cdot (l_1[n+1]), \delta_2''' \cdot l_2[m+1], n+1, m+1)$. The proof of this claim is done, as in the previous case, dispatching over (δ_1, δ_2) in $\mathcal{A}^{l_1,l_2}_{n+1,m+1}$.

ip_2. $\mathcal{E}(\delta_1'' \cdot l_1[n+1], \delta_2'' \cdot (0:)) \not\leq \mathcal{E}(\delta_1' \cdot (0:), \delta_2' \cdot l_2[m+1])$: Then there exists only two cases:

$\mathsf{ip}_{2.1}$. $\mathcal{E}(\delta_1''' \cdot l_1[n+1], \delta_2''' \cdot l_2[m+1]) \leq \mathcal{E}(\delta_1'' \cdot l_1[n+1], \delta_2'' \cdot (0:))$: We claim $Q(\delta_1'' \cdot l_1[n+1], \delta_2'' \cdot (0:), n+1, m+1)$.
$\mathsf{ip}_{2.2}$. $\mathcal{E}(\delta_1''' \cdot l_1[n+1], \delta_2''' \cdot l_2[m+1]) \not\leq \mathcal{E}(\delta_1'' \cdot l_1[n+1], \delta_2'' \cdot (0:))$: We claim $Q(\delta_1''' \cdot l_1[n+1], \delta_2''' \cdot l_2[m+1], n+1, m+1)$. The proofs of the previous two claims is done dispatching over (δ_1, δ_2) in $\mathcal{A}^{l_1,l_2}_{n+1,m+1}$. □

The theorem 3.1 can be simply modified in order to construct not only the alignment with highest score but also the score itself (that is the *similarity*).

The program extracted from the previous proof is the following:

```
[l,g,alpha]
(Rec nat=>nat=>(list nat @@ list nat))
 ([m] if (m=0) ((:), (:))
               ((zeros (m+1)), (pre (m+1) g))
 ([n, f: (nat=>(list nat @@ list nat))]
  (Rec nat=>(list nat, list nat))
   ((pre (n+1) l), (zeros (n+1)))
   ([m,(d_1',d_2')]
     [LET (d_1'', d_2'') = (f (m+1)) IN
      [LET (d_1''', d_2''') = (f m) IN
        [IF ((E (d_1'':+: l[n+1]) 0 alpha) <=
              (E (d_1':+: (:)) (d_2':+: g[m+1]) alpha))
         [IF ((E (d_1''':+: l[n+1]) (d_2''':+: g[m+1]) alpha)
               <=(E (d_1':+: (:)) (d_2':+: g[m+1]) alpha))
           ((d_1':+:(:)), (d_2':+: g[m+1]))
           ((d_1''':+: l[n+1]) (d_2''':+: g[m+1]))]
         [IF ((E (d_1''':+: l[n+1]) (d_2''':+: g[m+1]) alpha)
               <=(E (d_1'':+: l[n+1]) (d_2'':+:  (:)) alpha))
           ((d_1'':+: l[n+1]) (d_2'' :+: (:)))
           ((d_1''':+:l[n+1]) (d_2''':+:g[m+1]))]]]])))|l||g|
```

Here we indicated by `(pre n)` the operator pre_n, by `(zeros n)` the string 0^n, by `E` the function $\mathcal{E}$ and by `alpha` the scoring function α.

Complexity of the Extracted Algorithm: The complexity of the extracted program can be modeled by the following recurrence:

$$T_1(n,m) = \begin{cases} k_1 m & n = 0 \\ T_2(m) & n > 0 \end{cases}$$

with

$$T_2(m) = \begin{cases} k_2 n & m = 0 \\ \quad T_2(m-1) + T_1(n-1,m) + \\ \quad T_1(n-1,m-1) + 2k_3\mathsf{max}(n+m) & m > 0 \end{cases}$$

Here k_1, k_2, k_3 are general constants and $2k_3\mathsf{max}(n+m)$ is the cost for the application of the append operation in the body of the nested recursion. The complexity of the extracted program then will be given by $T_1(|l_1|, |l_2|)$. Given $n > 0$ and $m > 0$ the unfolding of $T_1(n,m)$ can be represented as a ternary tree where the lowest branch has high m and the highest $n+m$. Thus the extracted programs has a number of recursive calls in $\Omega(3^m) \cap \Omega(3^{n+m})$.

4. LIST AS MEMORY PARADIGM

To drastically reduce the complexity of our extracted program, we developed a method that we named *list as memory*. The idea consist in evaluating a sufficient amount of data in advance so that the extracted algorithm gets to reuse it instead of recomputing it each time it is needed.

The basic idea is still to prove Theorem3.1 by a double induction (before on the length $|l_1|$ of the first list and by a nested induction on $|l_2|$ length of the second list) but this time using an additional data structure w, a FIFO (F*irst* I*n* F*irst* O*ut*) list where we store the alignments with highest score computed in the previous steps. The list w will be built and updated during the proof and it will constitutes part of the witness of the new proof together with the alignment of highest score.

Thus assuming we want to compute the best alignment of the first $n+1$ characters of l_1 and $m+1$ character of l_2, we will assume w to be the following list of alignments:

$$(\delta_1,\delta_2)^{l_1,l_2}_{n,m},(\delta_1,\delta_2)^{l_1,l_2}_{n,m+1},\ldots,(\delta_1,\delta_2)^{l_1,l_2}_{n,|l_2|},\\ (\delta_1,\delta_2)^{l_1,l_2}_{n+1,0},(\delta_1,\delta_2)^{l_1,l_2}_{n+1,1},\ldots,(\delta_1,\delta_2)^{l_1,l_2}_{n+1,m}$$

with $(\delta_1,\delta_2)^{l_1,l_2}_{i,j}$ alignment of highest score between the first i characters of l_1 and j characters of l_2. At this point the intended alignment it will be computed considering the head of w, $(\delta_1,\delta_2)^{l_1,l_2}_{n,m}$, the head of the tail of w, $(\delta_1,\delta_2)^{l_1,l_2}_{n,m+1}$ and the recursive call of the nested induction on l_2 (the alignment of highest score between the first $n+1$ element of l_1 and m elements of l_2, that here occur as last element in w) Once the new alignment is computed the list w has to be properly updated.

So in general the idea is to replace the double instantiation of the induction hypothesis (2) in Theorem 3.1 (that correspond to the two recursive calls in the extracted algorithm) with a reading operation of the *head* and the *head* of the *tail* of our *memory* list w.

In order to use such memory list in our proof we have to modify in an appropriate way the original proof of the Theorem 3.1. More precisely we introduce the predicate $\texttt{MEM} \subseteq \mathbf{L}(\mathbf{N}_{>0})\times\mathbf{L}(\mathbf{N}_{>0})\times\mathbf{N}\times\mathbf{N}\times\mathbf{L}(\mathbf{L}(N)\times\mathbf{L}(N))$ where,

- $(\texttt{MEM}\ l_1\,l_2\,0\,v\,w)$, stands for "in w are stored the the $v+1$ alignments $(0^k, pre_k(l_2))$ with $k=0,\ldots v$" (here we assume $0^0=(:)$ and $pre_0(l_2)=(:)$) and
- $(\texttt{MEM}\ l_1\,l_2\,(u+1)\,v\,w)$, stands for "in w are stored the $|l_2|+2$ alignments of highest scores between the first j and k characters of l_1 and l_2 with

  $$(j,k)\in\{(u,v),\ldots,(u,|l_2|),(u+1,0),\ldots,(u+1,v)\}$$

and the following set of axioms specifying the necessary operations to build and correctly update the memory list w:

[I](*Initialization*),

$$\forall l_1,l_2,m,w(\texttt{MEM}\ l_1\,l_2\,0\,m\,(\mathsf{init}\,m\,l_2))$$

with

$$(\mathsf{init}\,m\,l_2)=\begin{cases}((:),(:)) & m=0\\ ((\mathsf{init}\,(m-1)):+:(0^m, pre_m\,l_2)) & 0<m\end{cases}$$

[H] (*Head of the list*):

$$\forall l_!,l_2,n,m,w(\texttt{MEM}\ l_1\,l_2\,(n+1)\,m\,w)\to\\ Q(\pi_0(\mathsf{head}w),\pi_1(\mathsf{head}w),n,m)$$

[HT] (*Head of the tail*):

$$\forall l_!,l_2,n,m,w(m<|l_2|)\to(\texttt{MEM}\ l_1\,l_2\,(n+1)\,m\,w)\to\\ Q(\pi_0(\mathsf{head}(\mathsf{tail}\,w)),\pi_1(\mathsf{head}(\mathsf{tail}\,w)),n,m+1)$$

[CL] (*Change Line*):

$$\forall l_1,l_2,n,m,w(\texttt{MEM}\ l_1\,l_2\,n\,|l_2|\,w)\to\\ (\texttt{MEM}\ l_1\,l_2\,(n+1)\,0\,((\mathsf{tail}\,w):+:((pre_{n+1}\,l_1),0^{n+1}))$$

[OSOR1] (*One Step On the Right 1*):

$$\forall l_1,l_2,n,m,\delta_1',\delta_2',w(m<|l_2|)\to(Q\,\delta_1'\,\delta_2'\,n\,m)\to\\ (\texttt{MEM}\ l_1\,l_2\,n\,m\,w)\to\\ (\mathcal{E}(\delta_1''\cdot l_1[n+1],\delta_2''\cdot(0:))\le\mathcal{E}(\delta_1'\cdot(0:),\delta_2'\cdot l_2[m+1]))\to\\ (\mathcal{E}(\delta_1'''\cdot l_1[n+1],\delta_2'''\cdot l_2[m+1])\le\mathcal{E}(\delta_1'\cdot(0:),\delta_2'\cdot l_2[m+1]))\to\\ (\texttt{MEM}\ l_1\,l_2\,n\,(m+1)\,((\mathsf{tail}\,w)\cdot(\delta_1'\cdot(0:),\delta_2'\cdot l_2[m+1])))$$

with $(\delta_1''',\delta_2''')=(\mathsf{head}\,w)$ and $(\delta_1'',\delta_2'')=(\mathsf{head}\,(\mathsf{tail}\,w))$.

[OSOR2] (*One Step On the Right 2*):

$$\forall l_1,l_2,n,m,\delta_1',\delta_2',w(m<|l_2|)\to(Q\,\delta_1'\,\delta_2'\,n\,m)\to\\ (\texttt{MEM}\ l_1\,l_2\,n\,m\,w)\to\\ (\mathcal{E}(\delta_1''\cdot l_1[n+1],\delta_2''\cdot(0:))\le\mathcal{E}(\delta_1'\cdot(0:),\delta_2'\cdot l_2[m+1]))\to\\ (\mathcal{E}(\delta_1'''\cdot l_1[n+1],\delta_2'''\cdot l_2[m+1])\not\le\mathcal{E}(\delta_1'\cdot(0:),\delta_2'\cdot l_2[m+1]))\to\\ (\texttt{MEM}\ l_1\,l_2\,n\,(m+1)\,((\mathsf{tail}\,w)\cdot(\delta_1'''\cdot l_1[n+1],\delta_2'''\cdot l_2[m+1])))$$

with $(\delta_1''',\delta_2''')=(\mathsf{head}\,w)$ and $(\delta_1'',\delta_2'')=(\mathsf{head}\,(\mathsf{tail}\,w))$.

[OSOR3] (*One Step On the Right 3*):

$$\forall l_1,l_2,n,m,\delta_1',\delta_2',w(m<|l_2|)\to(Q\,\delta_1'\,\delta_2'\,n\,m)\to\\ (\texttt{MEM}\ l_1\,l_2\,n\,m\,w)\to\\ (\mathcal{E}(\delta_1''\cdot l_1[n+1],\delta_2''\cdot(0:))\not\le\mathcal{E}(\delta_1'\cdot(0:),\delta_2'\cdot l_2[m+1]))\to\\ (\mathcal{E}(\delta_1'''\cdot l_1[n+1],\delta_2'''\cdot l_2[m+1])\le(\delta_1''\cdot l_1[n+1],\delta_2''\cdot(0:)))\to\\ (\texttt{MEM}\ l_1\,l_2\,n\,(m+1)\,((\mathsf{tail}\,w)\cdot(\delta_1''\cdot l_1[n+1],\delta_2''\cdot(0:))))$$

with $(\delta_1''',\delta_2''')=(\mathsf{head}\,w)$ and $(\delta_1'',\delta_2'')=(\mathsf{head}\,(\mathsf{tail}\,w))$.

[OSOR4] (*One Step On the Right 4*):

$$\forall l_1,l_2,n,m,\delta_1',\delta_2',w(m<|l_2|)\to(Q\,\delta_1'\,\delta_2'\,n\,m)\to\\ (\texttt{MEM}\ l_1\,l_2\,n\,m\,w)\to\\ (\mathcal{E}(\delta_1''\cdot l_1[n+1],\delta_2''\cdot(0:))\not\le\mathcal{E}(\delta_1'\cdot(0:),\delta_2'\cdot l_2[m+1]))\to\\ (\mathcal{E}(\delta_1'''\cdot l_1[n+1],\delta_2'''\cdot l_2[m+1])\not\le(\delta_1''\cdot l_1[n+1],\delta_2''\cdot(0:)))\to\\ (\texttt{MEM}\ l_1\,l_2\,n\,(m+1)\,((\mathsf{tail}\,w)\cdot(\delta_1'''\cdot l_1[n+1],\delta_2'''\cdot l_2[m+1])))$$

with $(\delta_1''',\delta_2''')=(\mathsf{head}\,w)$ and $(\delta_1'',\delta_2'')=(\mathsf{head}\,(\mathsf{tail}\,w))$.

THEOREM 4.1. ***[I]*** $\to$ ***[CL]*** $\to$ ***[H]*** $\to$ ***[HT]*** $\to$ ***[OSOR1]*** $\to$ ***[OSOR2]*** $\to$ ***[OSOR3]*** $\to$ ***[OSOR4]*** $\to\forall l_1,l_2(\exists\delta_1,\delta_2(\delta_1,\delta_2)\in\mathcal{A}^{l_1,l_2}_{|l_1|,|l_2|}\wedge\forall\delta_1',\delta_2'((\delta_1',\delta_2')\in\mathcal{A}^{l_1,l_2}_{|l_1|,|l_2|}\to\mathcal{E}(\delta_1',\delta_2')\le\mathcal{E}(\delta_1,\delta_2)))\wedge$

$\exists w(\mathit{MEM}\; l_1\, l_2\, |l_1|\, |l_2|\, w)$

Proof Sketch. Assume **[I]**, **[CL]**, **[H]**, **[HT]**, **[OSOR1]**, **[OSOR2]**, **[OSOR3]**, **[OSOR4]** l_1 and l_2. In order to prove the thesis we prove the following assertion:

$$\forall n, m(\exists \delta_1, \delta_2(\delta_1, \delta_2) \in \mathcal{A}^{l_1,l_2}_{n,m} \wedge \forall \delta'_1, \delta'_2(\delta'_1, \delta'_2) \in \mathcal{A}^{l_1,l_2}_{n,m} \to \mathcal{E}(\delta'_1, \delta'_2) \leq \mathcal{E}(\delta_1, \delta_2)) \wedge \exists w(\texttt{MEM}\; l_1\, l_2\, n\, m\, w)$$

By induction on n and m.
Base Case$[n = 0]$ We prove

$$\forall m \exists \delta_1, \delta_2 Q(\delta_1, \delta_2, 0, m) \wedge \exists w(\texttt{MEM}\; l_1\, l_2\, 0\, m\, w)$$

by case distinction over m:
Base Case$[n = 0,\, m = 0]$
$Q((:), (:), 0, 0) \wedge \exists w(\texttt{MEM}\; l_1\, l_2\, 0\, 0\, (\mathsf{init}\, 0\, l_2)))$ by rule (A_0) and **[I]**.
Induction Step$[n = 0,\, m + 1]$
We have $Q(0^{m+1}, pre_{m+1}(l_2), 0, m + 1) \wedge$
$\exists w(\texttt{MEM}\; l_1\, l_2\, 0\, (m + 1)\, (\mathsf{init}\, (m + 1)\, l_2))$ by rule (A_1) and **[I]**.

Induction Step$[n + 1]$
We now assume

$$\forall m \exists \delta_1, \delta_2 Q(\delta_1, \delta_2, n, m) \wedge \exists w(\texttt{MEM}\; l_1\, l_2\, n\, m\, w) \qquad (7)$$

and we show

$$\forall m \exists \delta_1, \delta_2 Q(\delta_1, \delta_2, n + 1, m) \wedge \exists w(\texttt{MEM}\; l_1\, l_2\, (n + 1)\, m\, w)$$

By induction over m:

Base Case$[n + 1,\, m = 0]$ $Q(pre_{n+1}(l_1), 0^{n+1}, n + 1, 0)$ by (A_2). Then instantiating (7) on $|l_2|$ we have $\overline{w}$ such that $\exists w(\texttt{MEM}\; l_1\, l_2\, n\, |l_2|\, \overline{w})$ and by **[CL]** we have $(\texttt{MEM}\; l_1\, l_2\, (n + 1)\, 0\, ((\mathsf{tail}\, \overline{w}) \cdot ((pre_{n+1}\, l_1), 0^{n+1}))$.

Induction Step$[n + 1,\, m + 1]$ Assume

$$\exists \delta_1, \delta_2 Q(\delta_1, \delta_2, n + 1, m) \wedge \exists w(\texttt{MEM}\; l_1\, l_2\, (n + 1)\, m\, w) \qquad (8)$$

we prove

$$\exists \delta_1, \delta_2 Q(\delta_1, \delta_2, n+1, m+1) \wedge \exists w(\texttt{MEM}\; l_1\, l_2\, (n+1)\, (m+1)\, w)$$

By (8) there exists δ'_1,δ'_2 such that $(\delta'_1, \delta'_2) \in \mathcal{A}^{l_1,l_2}_{n+1,m}$ and such that for every $(\delta_1, \delta_2) \in \mathcal{A}^{l_1,l_2}_{n+1,m}$

$$\mathcal{E}(\delta_1, \delta_2) \leq \mathcal{E}(\delta'_1, \delta'_2) \qquad (9)$$

By (8) let $\overline{w}$ be such that $(\texttt{MEM}\; l_1\, l_2\, (n+1)\, m\, \overline{w})$. By **[HT]**, we have that $(\delta''_1, \delta''_2) \in \mathcal{A}^{l_1,l_2}_{n,m+1}$ and for every $(\delta_1, \delta_2) \in \mathcal{A}^{l_1,l_2}_{n,m+1}$

$$\mathcal{E}(\delta_1, \delta_2) \leq \mathcal{E}(\delta''_1, \delta''_2) \qquad (10)$$

with $(\delta''_1, \delta''_2) = (\mathsf{head}(\mathsf{tail}\, \overline{w}))$.

By **[H]** we have that that $(\delta'''_1, \delta'''_2) \in \mathcal{A}^{l_1,l_2}_{n,m}$ and for every $(\delta_1, \delta_2) \in \mathcal{A}^{l_1,l_2}_{n,m}$

$$\mathcal{E}(\delta_1, \delta_2) \leq \mathcal{E}(\delta'''_1, \delta'''_2) \qquad (11)$$

with $(\delta''_1, \delta''_2) = (\mathsf{head}\, \overline{w})$. Now we have to dispatch over the following cases:

ip_1.$\mathcal{E}(\delta''_1 \cdot l_1[n + 1], \delta''_2 \cdot (0 :)) \leq \mathcal{E}(\delta'_1 \cdot (0 :), \delta'_2 \cdot l_2[m + 1])$: Then, only 2 cases are possible:

$\mathsf{ip}_{1.1}$. $\mathcal{E}(\delta'''_1 \cdot l_1[n + 1], \delta'''_2 \cdot l_2[m + 1]) \leq \mathcal{E}(\delta'_1 \cdot (0 :), \delta'_2 \cdot l_2[m + 1])$: We claim

$$Q(\delta'_1 \cdot (0 :), \delta'_2 \cdot l_2[m + 1], n + 1, m + 1)$$

and

$$(\texttt{MEM}\; l_1\, l_2\, n\, (m + 1)\, ((\mathsf{tail}\, w) \cdot (\delta'_1 \cdot (0 :), \delta'_2 \cdot l_2[m + 1])))$$

This is proved dispatching over (δ_1, δ_2) in $\mathcal{A}^{l_1,l_2}_{n+1,m+1}$ and by **[OSOR1]**, ip_1 and $\mathsf{ip}_{1.1}$.

$\mathsf{ip}_{1.2}$. $\mathcal{E}(\delta'''_1 \cdot l_1[n + 1], \delta'''_2 \cdot l_2[m + 1]) \not\leq \mathcal{E}(\delta'_1 \cdot (0 :), \delta'_2 \cdot l_2[m + 1])$: We claim

$$Q(\delta'''_1 \cdot (l_1[n + 1]), \delta'''_2 \cdot l_2[m + 1], n + 1, m + 1)$$

and

$$(\texttt{MEM}\; l_1\, l_2\, n\, (m + 1)\, ((\mathsf{tail}\, w) \cdot (\delta'''_1 \cdot l_1[n + 1], \delta'''_2 \cdot l_2[m + 1])))$$

This is proved dispatching over (δ_1, δ_2) in $\mathcal{A}^{l_1,l_2}_{n+1,m+1}$ and by **[OSOR2]**, ip_1 and $\mathsf{ip}_{1.2}$

ip_2. $\mathcal{E}(\delta''_1 \cdot l_1[n + 1], \delta''_2 \cdot (0 :)) \not\leq \mathcal{E}(\delta'_1 \cdot (0 :), \delta'_2 \cdot l_2[m + 1])$:Then there exists only two cases:

$\mathsf{ip}_{2.1}$. $\mathcal{E}(\delta'''_1 \cdot l_1[n + 1], \delta'''_2 \cdot l_2[m + 1]) \leq \mathcal{E}(\delta''_1 \cdot l_1[n + 1], \delta''_2 \cdot (0 :))$: We claim

$$Q(\delta''_1 \cdot l_1[n + 1], \delta''_2 \cdot (0 :), n + 1, m + 1)$$

and

$$(\texttt{MEM}\; l_1\, l_2\, n\, (m + 1)\, ((\mathsf{tail}\, w) \cdot (\delta''_1 \cdot l_1[n + 1], \delta''_2 \cdot (0 :)))$$

Proved dispatching over (δ_1, δ_2) in $\mathcal{A}^{l_1,l_2}_{n+1,m+1}$ and by **[OSOR3]**, ip_2 and $\mathsf{ip}_{2.1}$

$\mathsf{ip}_{2.2}$. $\mathcal{E}(\delta'''_1 \cdot l_1[n + 1], \delta'''_2 \cdot l_2[m + 1]) \not\leq \mathcal{E}(\delta''_1 \cdot l_1[n + 1], \delta''_2 \cdot (0 :))$: We claim

$$Q(\delta'''_1 \cdot l_1[n + 1], \delta'''_2 \cdot l_2[m + 1], n + 1, m + 1)$$

and

$$(\texttt{MEM}\; l_1\, l_2\, n\, (m + 1)\, ((\mathsf{tail}\, w) \cdot (\delta'''_1 \cdot l_1[n + 1], \delta'''_2 \cdot l_2[m + 1]))$$

Proved dispatching over (δ_1, δ_2) in $\mathcal{A}^{l_1,l_2}_{n+1,m+1}$ and by proved by **[OSOR4]**, ip_2 and $\mathsf{ip}_{2.2}$ □

From the previous proof we can extract the following program:

```
[l,g,alpha]
(Rec nat=>nat=>((list nat@@list nat)@@
                   (list(list nat@@list nat))
 ([m] [if (m=0)
      (((:),(:)) , ((:),(:)))
      (((nZeros (m+1)), (nPrefix (m+1) g)),(init (m+1) g)


([n,f:(nat=>((list nat@@list nat)@@
                        (list(list nat@@list nat)))]
  (Rec nat=>((list nat@@list nat)@@
                         (list(list nat@@list nat)))
   LET w = (f |g|) IN
    (((nPrefix (n+1) l), (nZeros (n+1))) ,
       ((tail w):+:((nPrefix (n+1) l),(nZeros(n+1)))))
    ([m,((d_1',d_2'),w)]
```

```
[LET (d_1'', d_2'') = (head (tail w)) IN
 [LET (d_1''', d_2''') = (head w) IN
   [IF((E (d_1'':+: l[n+1]) 0 alpha) <=
        (E (d_1':+: (:)) (d_2':+: g[m+1]) alpha))
    [IF((E(d_1''':+: l[n+1])(d_2''':+: g[m+1])alpha)
         <=(E (d_1':+: (:))(d_2':+: g[m+1]) alpha))
     (((d_1':+:(:)),(d_2':+:g[m+1])),
       ((tail w):+:(d_1':+:(:)),(d_2':+:g[m+1])))
     (((d_1''':+: l[n+1])(d_2''':+:g[m+1])),
      ((tail w):+:(d_1''':+: l[n+1])(d_2''':+:g[m+1])))]
    [IF((E(d_1''':+: l[n+1]) (d_2''':+: g[m+1])alpha)
         <= (E(d_1'':+:l[n+1]) (d_2'':+: (:))alpha))
     (((d_1'':+: l[n+1])(d_2'' :+: (:))),
       ((tail w):+:((d_1'':+: l[n+1]) (d_2'':+:(:)))))
     (((d_1''':+: l[n+1]) (d_2''':+: g[m+1])),
      ((tail w):+:((d_1''':+:l[n+1]) (d_2''':+:g[m+1]))))
     ]]]])))|l| |g|
```

4.0.4 Complexity Considerations

The complexity of the extracted program can be modeled by the following recurrence (here we have as additional parameter the length of g):

$$T_1(n,m) = \begin{cases} k_1 m & n = 0 \\ T_2(n,m) & n > 0 \end{cases}$$

with

$$T_2(n,m) = \begin{cases} T_1(n-1,|g|) & m = 0 \\ T_2(n,m-1) + 2k_2\mathsf{max}(n+m) & m > 0 \end{cases}$$

Here k_1 and k_2 are general constants. Given $|l| > 0$ and $|g| > 0$ the unfolding of $T_1(|l|,|g|)$ can be represented by the following $|l| \times |g|$ matrix of list of calls:

$$\begin{array}{llll} T_1(|l|,|g|) & \to T_2(|l|,|g|) & \to \ldots \to & T_2(|l|,0) \\ \to T_1(|l|-1,|g|) & \to T_2(|l|-1,|g|) & \to \ldots \to & T_2(|l|-1,0) \\ & \vdots & & \\ \to T_1(1,|g|) & \to T_2(1,|g|) & \to \ldots \to & T_2(1,0) \end{array}$$

and being the complexity of each call $2k_2\mathsf{max}(|l|+|g|)$ then $T_1(|l|,|g|)$ is in $\mathcal{O}(|l||g|\mathsf{max}(|l|+|g|))$

5. CONCLUSIONS

With an opportune modification of the alignment definition in Figure 3 we can avoid the cost relative to the applications of the append function. In this way, the extracted program from the efficient implementation of the existence of an alignment with highest score will have a complexity in $\mathcal{O}(|l||g|)$. Future work will regards a sort of automation of the presented method.

6. REFERENCES

[1] Dan Gusfield. *Algorithms on Strings, Tree and Sequences*. Cambridge University Press, 1997.

[2] M.H. Sørensen and P.Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 2006.

[3] G. Kreisel. Interpretation of Analysis by means of Functionals of Finite Type. In A. Heyting, editor, *Constructivity in Mathematics*, 1959.

[4] Helmut Schwichtenberg. Minimal logic for computable functionals. February 2008.

Model-Checking Modulo Theories at Work: the integration of Yices in MCMT

Silvio Ghilardi
Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano
ghilardi@dsi.unimi.it

Silvio Ranise
Dipartimento di Informatica
Università degli Studi di Verona
silvio.ranise@univr.it

ABSTRACT
Recently, the notion of an array-based system has been introduced as an abstraction of infinite state systems (such as parametrised systems) which allows for model checking safety properties by SMT solving. Unfortunately, the use of quantified first-order formulae to describe sets of states makes checking for fix-point and unsafety extremely expensive. In this paper, we describe (static and dynamic) techniques to overcome this problem which have been implemented in the (declarative) model checker MCMT. We describe how such techniques have been combined with Yices (the back-end SMT solver) and discuss some interesting experimental results.

Keywords
Declarative model checking, Backward reachability, SMT, Yices

1. INTRODUCTION
Safety properties for classes of infinite state systems can be checked by a (backward) reachability analysis, i.e. repeatedly computing the pre-image of the set of unsafe states (obtained by complementing the property to be verified) and checking for fix-point and emptiness of the intersection with the set of initial states. This analysis is a decision procedure for some classes of parametrised systems [1], i.e. systems consisting of a finite (but unknown) number n of identical processes which can be modelled as extended finite state automata, manipulating variables whose domains can be unbounded, like integers. The challenge is to verify properties for any number n of processes.

Backward reachability analysis is parametrised with respect to the topology of the parametrised system and the domains of the variables manipulated by each process in the system. Sometimes, the topology of the system can be safely abstracted away (e.g., by counting abstraction [7]) so as to obtain a guarded transition systems where only numeric variables are used. Systems based on constraint solving techniques capable of handling this kind of transition systems have been successful for some classes of systems (see, e.g., again [7]). However, for many parametrised, counting abstractions are not precise and spurious counter-examples are introduced. To overcome this problem, there have been attempts at designing 'topology-aware' backward reachability analyses (see, e.g., [2, 3, 4]). Usually, these approaches encodes the topology in a suitable data-structure which allows for the finite representation of an infinite set of states of the parametrised systems while the data manipulated by each process in the system are handled declaratively by (numerical) constraints. These approaches are hybrid as they exploit both dedicated data structures and (declarative) constraints to represent the infinite sets of states of the system. This implies that the computation of the pre-image also requires manipulations of constraint expressions for the data variables and of the structure encoding the topology of the system. So, while manipulations on constraints depends only on very general properties of the constraint structure itself and can thus be re-used without much effort, this is not the case for the topology which is represented by *ad hoc* data structures. In fact, every time the topology changes, the computation of the pre-image must be designed from scratch and the requirements ensuring the termination of the backward analysis must be checked; e.g., the existence of a suitable pre-order (i.e. it is a reflexive and transitive binary relation) on configurations which finitely represent infinite set of states (see [1] for details).

In [9], to overcome all these problems and obtain a fully declarative approach to backward reachability of infinite state systems, we proposed the notion of *array-based systems* as a suitable abstraction for parametrised systems as well as sequential programs manipulating arrays, or lossy channel systems. The idea is to use classes of first-order formulae to represent an infinite set of states of the system so as to simplify the computation of pre-images. As it is standard in deductive verification of software, arrays are modelled by functions. In this framework, both the topology and the data manipulated by the systems are specified by using a suitable class of first-order structures, which is modularly and uniformly obtained by combining a class of structures for the topology and one for the data via the functions representing arrays. More precisely, both the topology and the data of the system are specified by first-order theories, which are pairs formed by a (first-order) language and a class of (first-order) structures. In this way, the union of the lan-

guages augmented with the function symbols representing arrays yields the language of the formulae representing the states of the system.

In order to mechanize the backward reachability analysis using formulae to represent states, three requirements must be satisfied. First, the class of formulae used to represent states must be closed under pre-image computation. Second, the tests for fix-point and safety should be reduced to decidable logical problems. Third, sufficient conditions on the theories specifying the topology and the data must be identified so as to guarantee termination of the backward reachability analysis. All these requirements have been investigated in our previous works [9, 12] and the interested is pointed to those papers for details. In particular, we recall that the computation of the pre-image for the class of guarded assignment systems consists of simple manipulations of formulae and that it is straightforward to show closure under pre-image computation [12]. We also recall how it is possible to reduce the checks for fix-point and safety to Satisfiability problems Modulo the Theories (SMT) for the topology and the data of the system of first-order formulae containing (universal) quantifiers. Under suitable hypotheses on the theories, this SMT problem turns out to be decidable [9, 12]. However, this is not yet enough to ensure the termination of the backward reachability analysis: in [9], we showed how to introduce a pre-order on the set of configurations of the system and observed that termination is achieved when such a pre-order is a well-quasi-order[1] (this is the case, among others, of broadcast protocols and lossy channels systems).

In this paper, we focus on the pragmatics of our approach and discuss techniques that allowed us to implement a prototype tool, called MCMT,[2] which proved to be competitive with the state-of-the-art model checker (for parametrised systems) PFS [2]. The crucial problem is solving the SMT problem for fix-point and safety checking, because it is required to be able to handle quantified formulae which makes the off-the-shelf use of SMT solvers problematic. In fact, even when using classes of formulae with decidable satisfiability problem, currently available SMT solvers are not yet mature enough to efficiently discharge formulae containing (universal) quantifiers, despite the fact that this problem has recently attracted a lot of efforts (see, e.g., [6, 8, 5]). To alleviate this problem, we have designed a general decision procedure for the class of formulae satisfying the requirements above, based on quantifier instantiation and SMT solving [9]. Unfortunately, the number of instances required by the instantiation algorithm is still very large and preliminary experiments have shown poor performances. This fact together with the observation that the size of the formulae generated by the backward search algorithm grows very quickly demand a principled approach to the pragmatics of efficiently integrating SMT solvers in backward analysis.

More precisely, we describe *instance reduction* heuristics which allows us to reduce the number of instances to be considered for the tests of fix-point and non-empty intersection with the initial set of states. We consider two types of techniques: *static* and *dynamic*. We illustrate the former by showing how a careful classification of the transitions based on their shape allows us to reduce the number of (universally) quantified variables and consequently also the number of their instances. For dynamic instance reduction techniques, we will see how to *filter* instances of formulae which cannot contribute to show unsatisfiability by (computationally inexpensive) reasoning *modulo* certain theories of *enumerated data-types*. We also describe that during the exploration of the symbolic state space by backward reachability, we have observed a sort of "locality principle" for fix-point testing whereby using the more recently visited formulae representing the set of backward reachable states has proved to detect unsatisfiability earlier; we call this *chronological fix-point checking*.

Finally, we describe the general architecture of our model-checker and its main loop while emphasizing the central role played by the SMT solver Yices;[3] and discuss some experimental results about a heuristics that we have implemented in the latest release of the tool.

2. ARRAY-BASED SYSTEMS AND MCMT

To make the paper self-contained, we briefly recall the notion of array-based system and related safety problem. The basic ingredients of an array based system are a theory T_I (with related signature Σ_I) for describing the topology of a system (i.e. T_I is a theory specifying the structure of the identifiers, also called indexes in the following, of processes in a parametrized system) and another theory T_E (with related possibly multi-sorted signature Σ_E) for describing data (i.e. numerical values, program counter locations, etc.); the array-based system has *array variables* $a, b, \ldots$ for local data: these are function variables having the index sort as source sort and some data sort as target sort. For example, to specify a system which requires only to identify processes by their identifiers, T_I will be the theory of equality (without functions); to state that processes are arranged in a linear array, T_I is the theory of linear orders whose signature consists just of a binary relation $<$ which allows one to locate processes 'on the left' or 'on the right' of a given process. Indeed, most available SMT solvers provide support for these theories: the former can be handled by using (part of) a congruence closure algorithm while the latter by using (part of) a decision procedure for Linear Arithmetics. Since it is more common to use constraints to handle the content of data variables, we only mention that SMT solvers (and Yices in particular) offer support for several instances of T_E. In particular, the theory of enumerated datatype (i.e. the theory whose signature consists of only a finite set of constants which are constrained to be pairwise distinct and to name all the elements in the domain of any structure in the class of models) is particularly useful to model control locations, i.e. the states of the extended automaton representing the processes in the system.

Once T_I and T_E are fixed, we consider the tuple of function symbols representing all array variables manipulated by the system. For simplicity, here, we consider array-based system with only one array variable a. Then, the specification of

[1]A well-quasi-ordering $\leq$ on a set X is a quasi-ordering (i.e., a reflexive, and transitive binary relation) such that any infinite sequence of elements $x_0, x_1, x_2, \ldots$ from X contains an increasing pair $x_i \leq x_j$ with $i < j$.

[2]http://homes.dsi.unimi.it/~ghilardi/mcmt

[3]http://yices.csl.sri.com

an array-based system consists of a formula $I(a)$ describing the *initial* sets of states and a *transition* formula $\tau(a, a')$ relating actual a and updated a' array variables. A *safety* or reachability problem for the array based system $\mathcal{S} = (a, I, \tau)$ is a formula $U(a)$ specifying a set of states the system should not be able to reach starting from a state in I and firing τ finitely many times.

Formulae I, τ, U are subject to some syntactic restrictions [9, 12, 11] that guarantee that all safety and fix-point tests needed for a standard *backward reachability analysis* [1] can be reduced to quantifier-free SMT problems (modulo T_I and T_E) by instantiation, i.e. to the kind of tests that can be efficiently handled by state-of-the-art SMT solvers like Yices. In the following, we use two classes of formulae to describe states: *primitive differentiated* and $\forall^I$-formulae. The former is used to describe sets of unsafe state and consists of the existential closure of conjunctions of literals containing all the literals needed to say that quantified variables range over distinct indexes (see [11] for details). The following is an example of primitive differentiated formula:

$$\exists \mathtt{x}, \mathtt{y}.(\mathtt{x} \neq \mathtt{y} \wedge \mathtt{a}[\mathtt{x}] = q_2 \wedge \mathtt{a}[\mathtt{y}] = q_2)$$

saying that there exists two distinct processes in the system, identified by the identifiers `x` and `y` (which are constrained to be distinct by the literal $\mathtt{x} \neq \mathtt{y}$) such that the control location of the processes `x` and `y` is q_2 (so, if q_2 identifies the critical section, then the formula above specifies the violation of mutual exclusion). In other words, the formula describes all the configurations of the system where there exists two distinct processes which are both in control location q_2.

The class of $\forall^I$-formulae is used to describe the set of initial states or invariants of the system. For example, to say that all processes initially are in control location q_0, it is possible to use the formula: $\forall \mathtt{x}.\mathtt{a}[\mathtt{x}] = q_0$. Another example is the negation of the primitive differentiated formula above, i.e.

$$\forall \mathtt{x}, \mathtt{y}.((\mathtt{a}[\mathtt{x}] = q_2 \wedge \mathtt{a}[\mathtt{y}] = q_2) \Rightarrow \mathtt{x} = \mathtt{y})$$

describing the set of states of the system satisfying mutual exclusion.

For transition formulae $\tau(a, a')$, we use formulae that correspond to guarded assignment systems. An example of such formulae is the following:

$$\exists x.(\mathtt{a}[\mathtt{x}] = q_2 \wedge \forall \mathtt{j}.a'[j] = \textit{if } \mathtt{j} = \mathtt{x} \textit{ then } q_3 \textit{ else } q_0),$$

where the first conjunct is the *guard* and the second is the update. Notice that the update can be seen to model a broadcast action in a parametrised system: a process `x` which is in the control location q_2 moves to q_3 while all the other processes react by going to control location q_0.

2.1 An example specification in MCMT

To illustrate how to work with MCMT, we consider the verification of the key safety property of a protocol used to maintain coherence in a system with multiple (local) caches as used, for example, in commercial multiprocessor systems. In the following, we specify the Illinois cache coherence protocol (see, e.g., [7] for details) as an array-based system in MCMT.[4] Each cache in the protocol may have four possible control locations: `invalid`, `dirty`, `shared`, or `exclusive`. This is specified to MCMT by the following declarations:[5]

```
:smt (define-type locations (subrange 1 4))
```

The keyword `:smt` tells the model checker that what follows is a Yices expression and should be added to the current context of Yices. In this case, we introduce the new type symbol `locations` in the current Yices context whose values may range over the set of integers $\{1, 2, 3, 4\}$; where `1` encodes `invalid`, `2` encodes `dirty`, etc. If we use integers to identify the different copies of a cache, the state of the Illinois protocol can be naturally represented by an array mapping a finite sub-set of the integers to locations. This is declared as follows:

```
:local a locations
```

where the keyword `:local` tells MCMT that `a` is an array variable whose indexes are a finite sub-set of the integers and whose elements must be of type `locations`. This declaration also implicitly defines the theory T_E of the elements stored in the array as the theory of an enumerated data-type containing four distinct elements, identified by the numerals from `1` to `4`. At the beginning, all the caches in the Illinois cache coherence protocol are in the state `invalid`. The corresponding declaration in MCMT is the following:

```
:initial
:var x
:cnj (= a[x] 1)
```

The keyword `:initial` indicates that we are going to specify the set of initial states, `:var` specifies which (implicitly universally quantified) variable is allowed to occur in the formula, and `:cnj` says that the space separated list of formulae (in Yices syntax) is intended to be read conjunctively. The elements of the array variable `a` are accessed by using square brackets as in C: this is a peculiarity of the MCMT syntax. The logical reading of the piece of specification above is the following formula: $\forall \mathtt{x}.\mathtt{a}[\mathtt{x}] = \mathtt{1}$, where `x` is a variable of type `int`.

The negation of cache coherence (describing the set of unsafe states) is that there should not be any two local caches in `dirty` or one cache in `dirty` and another one in `shared`. This is specified in MCMT as follows:

```
:unsafe
:var x
:var y
:cnj (= a[x] 2) (> a[y] 1) (< a[y] 4)
```

[4]The file containing the complete specification of this example (with many others) is available from `http://homes.dsi.unimi.it/~ghilardi/mcmt`.

[5]The syntax of MCMT includes a sub-set of Yices syntax. Roughly, the rule is that all the keywords starting with a semicolon are specific to the model checker while the remaining should be valid expressions of the input language of Yices.

Indeed, the keyword `:unsafe` tells the tool that we are going to specify the set of unsafe states, the two `:var`'s introduce the (implicitly existentially quantified) variables which are allowed to occur in the formula, and furthermore the two variables are assumed to be mapped to two distinct integers (this is because, when describing sets of states, MCMT uses primitive differentiated formulae.The logical reading of this piece of specification is thus the following:

$$\exists \mathtt{x}, \mathtt{y}.(\mathtt{x} \neq \mathtt{y} \wedge \mathtt{a}[\mathtt{x}] = 2 \wedge \mathtt{a}[\mathtt{y}] > 1 \wedge \mathtt{a}[\mathtt{y}] < 4).$$

Notice that by using integers to encode locations, we are able to avoid the explicit use of disjunction as $\mathtt{a}[\mathtt{y}] > 1 \wedge \mathtt{a}[\mathtt{y}] < 4$ is logically equivalent to $\mathtt{a}[\mathtt{y}] = 2 \vee \mathtt{a}[\mathtt{y}] = 3$.

The full specification of the Illinois protocol requires eleven transitions to be specified [7]. For lack of space, we just give two of them:

```
:comment t1                 :comment t2
:transition                 :transition
:var x                      :var x
:var j                      :var j
:guard (= a[x] 1)           :guard (= a[x] 2)
:numcases 2                 :numcases 2
:case (= x j)               :case (= x j)
 :val 2                      :val 1
:case (not (= x j))         :case (not (= x j))
 :val 1                      :val a[j]
```

The transition on the left is a (so-called) 'write miss,' i.e. one cache (identified by `x`) gets an exclusive copy of the data and it moves to the control location `dirty` if it was `invalid`, whereas all others (identified by `j`) go to `invalid`. The logical reading of this piece of specification is the following:

$$\exists \mathtt{x}. \left(\begin{array}{l} \mathtt{a}[\mathtt{x}] = 1 \;\wedge \\ \forall \mathtt{j}.\mathtt{a}[\mathtt{j}] = (\;\textit{if}\; \mathtt{x} = \mathtt{j}\; \textit{then}\; 2\; \textit{else}\; 1) \end{array} \right),$$

where $\mathtt{a}[\mathtt{x}] = 1$ is called a *guard* (cf. the keyword `:guard`) and $\forall \mathtt{j}.\mathtt{a}[\mathtt{j}] = (\textit{if } \mathtt{x} = \mathtt{j} \textit{ then } 2 \textit{ else } 1)$ is called a *case-defined function* update (cf. the keyword `:case`). The values stored in the elements of the array `a` after the execution of the transition are specified after the keyword `:val`. (If there is more than one array variable, for each case of the function, the user is supposed to specify the values of each array variable, according to the order in which they were declared at the beginning of the input file.) In its current release, MCMT supports case-defined functions with at most 20 cases, specified by the `:numcases` keyword above. According to the logical reading, the variable `x` is implicitly existentially quantified while the variable `j` is implicitly universally quantified. And this is so, despite the fact that the two variables are introduced by the same keyword `:var`. To clarify this point, we must say that, in MCMT, transitions may have at most two existentially quantified variables and one universally quantified variable: the former are assumed to have `x` and `y` as identifiers while the latter to have `j` as identifier. This rigid convention is to simplify parsing as much as possible and it will be relaxed in future releases of the tool.

The transition on the right is called a 'replacement,' according to which the content of a `dirty` cache is written in the main memory: the cache gets `invalid`ated, whereas the remaining ones maintain their status. The logical reading of the transition on the right is given by following formula:

$$\exists \mathtt{x}. \left(\begin{array}{l} \mathtt{a}[\mathtt{x}] = 2 \;\wedge \\ \forall \mathtt{j}.\mathtt{a}[\mathtt{j}] = (\;\textit{if}\; \mathtt{x} = \mathtt{j}\; \textit{then}\; 1\; \textit{else}\; \mathtt{a}[\mathtt{j}]) \end{array} \right).$$

Since to specify the initial and final set of states, and all the transitions (also those not shown above), we have used only the equality symbol, the theory T_I over the indexes can be assumed to be the theory of equality. This is a declarative specification of the topology assumed by the cache coherence protocols to ensure the coherence property. In general, it is possible to characterize several different topologies for a distributed system by choosing a suitable theory T_I, such as the theory of linear orders for linearly ordered collections of processes, the theory of trees or graphs for the corresponding topologies. MCMT supports the declarative specifications of several different background theories both by using the `:smt` directive—as explained above—to enrich the Yices context with new type declarations and the keyword `:system_axiom` that allows one to extend the context with axioms concerning certain predicate symbols playing the role, for example, of recognizers for trees or graphs.

Two remarks are in order. First, because of the declarative approach underlying MCMT, alternative specifications of the same cache coherence protocol can be fed to the system. For example, those obtained by using counting abstraction [7] are easy to formalize in MCMT and Yices gives support to handle the resulting SMT problems which are modulo the theory of Linear Arithmetics. In the case of the Illinois protocol considered above, its counting abstraction is easily handled by MCMT. For details on the performances on this and other specifications, the reader is referred to Section 3. The second remark is about the input specification language of MCMT which is quite difficult to understand by humans. However, this should not be regarded as a drawback since the language has been designed for ease of parsing or as a target language of translators from richer and human-friendly specification languages.

2.2 More specification constructs

We consider here a couple of interesting constructs of MCMT specification language not exemplified by the (partial) specifications of the Illinois protocol given above.

Shared variables. An important mechanism to propagate information in distributed systems is to have shared variables that processes can read or update. In MCMT, it is possible to declare every such variable, say `g`, as follows:

```
:global g int
```

which means that `g` can be seen as a (single) integer variable. However, the system requires to dereference its value as a 'standard' array variable (declared by `:local`) such that

$$\forall \mathtt{x}, \mathtt{y}.\mathtt{g}[\mathtt{x}] = \mathtt{g}[\mathtt{y}], \tag{1}$$

i.e. `g` stores the same value in every one of its cells. In other words, every process in the system has a local copy of the shared variable whose value is identical to the value of the local copies of all the other processes. Indeed, transitions

must be designed in such a way to ensure that processes always have an identical copy of the same value. Internally, the tool exploits the information contained in the `:global` declaration by asserting the invariant (1) in the current Yices context.

Universal guards. In the two transitions of the (original) specification of the Illinois protocol, we have encountered only *existential* conditions, specified by the keyword **`:guard`**. However, there are other transitions of the protocol for which *universal* (also called, global) conditions are required. In general, this kind of transitions turns out to be useful for specifying some class of parametrised systems, i.e. distributed systems consisting of a finite collection of processes (see [13] for examples of this). To illustrate the notion of universal condition, consider a guard saying that a process i can execute a transition if a certain condition is satisfied by *all* processes $j \neq i$. MCMT supports also this kind of condition by the keyword **`:uguard`**. Internally, the tool automatically translates such universal guards into existential ones by recasting the parametrised system under consideration in the so-called *stopping failures model* [13], which is quite close to the *approximate* model of [3, 2]. The key property of a parametrized system modelled according to the stopping failures model is that processes may fail without warning at any time. To formalize this, we add q_{crash} to the set of control locations of each process in the system and a set of transitions saying that it is always possible to go from any control location to q_{crash}, for any process. Using q_{crash} it is possible to transform a universal into an existential guard at the expense of introducing more runs in the systems, thereby preserving safety and even some liveness properties such as recurrence (in practice, the system provides "real" safety certifications, but it can—quite rarely—give spurious traces of unsafety). To illustrate, consider the universal guard saying that a process i can execute a transition if a certain predicate C is satisfied by all processes $j \neq i$. In the stopping failures model, this can be expressed without the universal quantification as follows: the process i takes the transition without checking the predicate C and, concurrently, all processes $j \neq i$ not satisfying C move to the location q_{crash}; moreover, all processes $j \neq i$ satisfying C behave as originally prescribed. For more details on this issue, the interested reader is pointed to [12, 10].

3. ANATOMY OF MCMT

To solve safety problems, MCMT uses a refinement of 'backward reachability analysis' (see, e.g., [1]). This procedure repeatedly computes the pre-images of the set of unsafe states. The key peculiarity of MCMT—as anticipated by the specification of the Illinois protocol in the previous section—is the use of a *completely declarative approach* where (a) both the topology of and the elements manipulated by distributed systems are formalized by (first-order) theories, (b) sets of states are described by (a certain class of) logical formulae, namely disjunctions of primitive differentiated formulae, and (c) transitions are also described by (a certain class of) logical formulae, corresponding to guarded assignment systems. There are two main crucial consequences of this approach. First, let a be the array state variables of the system, $K(a)$ be a primitive differentiated formula describing the set of reachable states, and $\tau(a, a')$ be the formula describing one of the transitions in the system (representing a guarded assignment); the computation of the pre-image $Pre(\tau, K)$ of $K(a)$ with respect to $\tau(a, a')$ is greatly simplified. In fact, it is possible to show [12] that $Pre(\tau, K)$, i.e.

$$\exists a'.(\tau(a, a') \wedge K(a'))$$

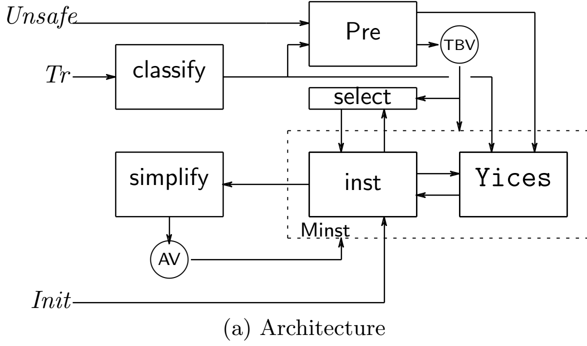
is logically equivalent to a disjunction of primitive differentiated formulae. The second important consequence of the declarative approach underlying MCMT is that checking for fix-point and intersection with the set of initial states can be reduced to a satisfiability problem modulo the combined theories T_I and T_E which are "connected" by the array a. The formulae to be checked for satisfiability are of the form $\exists \underline{i} \forall \underline{j} \psi(\underline{i}, \underline{j}, a[\underline{i}], a[\underline{j}])$, where $\underline{i}, \underline{j}$ are tuples of index variables and ψ is a quantifier free formula (notice that such formulae contain universally quantified variables in $\underline{j}$ and the existentially quantified variables in $\underline{i}$ may be regarded as Skolem constants). Under suitable hypotheses on T_I and T_E, it can be shown that the satisfiability of these formulae is decidable [9]. The proof of the decidability result suggests a two-phase procedure: (1) instantiate the universally quantified $\underline{j}$ to the Skolem constants $\underline{i}$ in all possible ways and (2) check the satisfiability of the resulting quantifier-free formulae modulo the theory obtained by the combination of T_I and T_E. Indeed, for efficiency, the two phases should be interleaved, the check for satisfiability can be done incrementally, and the satisfiability in the combination of T_I and T_E can be solved by using standard combination methods readily available in state-of-the-art SMT solvers. The instantiation phase is one of the main bottle-necks of the system and many of the heuristics implemented in MCMT have been developed to reduce the number of instantiations to be considered.

The main modules of MCMT and their connections are depicted in Figure 1(a) while its main loop can found in Figure 1(b). Notice the key role played by Yices (cf. Figure 1(a)), as (almost) all modules in the system invoke the available SMT solver. The integration of Yices in our model checker has been greatly simplified by the availability of the lightweight API. Particularly crucial to efficiently handle the SMT problems of quantified formulae for fix-point and safety checks is the incrementality of satisfiability checks so that the quantifier-free instances are incrementally added and unsatisfiability is reported as soon as possible. A nice set of interface functionalities is still missing from the SMT-Lib standardization effort.[6] Once available, this would enable us to perform interesting experimental investigations in using several SMT solvers by adopting the standard SMT-Lib language.

3.1 Main loop

The system takes as input the description of a safety problem consisting of the formulae describing the initial set of states ($Init$), its transitions (Tr), and the set of unsafe states ($Unsafe$). First of all, transitions are processed to extract relevant information for the successive computation of pre-images (module classify at lines 1–2). This is done in order to reduce the number of instantiated formulae in the satisfiability checks at lines 3, 9, and 10. Then, the formula $\exists \underline{i}.\forall \underline{j}. Unsafe(\underline{i}) \wedge Init(\underline{j})$ is checked for satisfiability

[6] `http://www.smt-lib.org/`



(a) Architecture

fun MCMT(*Init*, *Tr*, *Unsafe*)
1 $Tr^\ell \leftarrow \emptyset$;
2 **foreach** $\tau \in Tr$ **do** $Tr^\ell \leftarrow \mathsf{classify}(\tau) \cup Tr^\ell$;
3 **if** $\mathsf{inst}(Init, \mathsf{getinds}(Unsafe), Unsafe) = \mathsf{sat}$
then return unsafe;
4 $P \leftarrow Unsafe$; $\mathsf{AV} \leftarrow \emptyset$; $\mathsf{TBV} \leftarrow \emptyset$;
5 **do**
6 $\mathsf{AV} \leftarrow \mathsf{AV} \cup P$;
7 **foreach** $\tau^\ell \in Tr^\ell$ **do**
8 $N \leftarrow \mathsf{simplify}(\mathsf{Pre}(\tau^\ell, P))$;
9 **if** $\mathsf{Minst}(\mathsf{AV} \cup \mathsf{TBV}, \mathsf{getinds}(N), N) = \mathsf{sat}$
then begin $\mathsf{TBV} \leftarrow \mathsf{TBV} \cup N$;
10 **if** $\mathsf{inst}(Init, \mathsf{getinds}(N), N) = \mathsf{sat}$
then return unsafe;
end
11 **end**
12 **if** $\mathsf{TBV} \neq \emptyset$ **then** $P \leftarrow \mathsf{select}(\mathsf{TBV})$;
13 **while** $\mathsf{TBV} \neq \emptyset$;
14 **return** safe;

(b) Main loop

Figure 1: High-level description of mcmt

(line 3). The module inst implements the two-phase decision procedure for formulae of the form $\exists \underline{i} \forall \underline{j} \psi(\underline{i}, \underline{j}, a[\underline{i}], a[\underline{j}])$ sketched above; it takes the formulae *Init* and *Unsafe* as input together with the set of (existentially quantified) index variables occurring in *Unsafe* (computed by the function getinds); inst uses Yices to check the satisfiability of the resulting quantifier-free formulae in the combination of the theories over indexes and elements (see Figure 1(a)). If the satisfiability check is positive, then the intersection between the set of initial states and that of unsafe states is non-empty and the array-based system is trivially unsafe (line 3). Otherwise (lines 4–13), the main part of the algorithm for backward reachability analysis is entered. The loop maintains two sets AV and TBV of primitive differentiated formulae; the former stores the sets of states which have been already visited while the latter those which are still to be visited (see the two circles in Figure 1(a)). The variable P contains the set of states which is currently under consideration; when entering the loop for the first time, this is the set of unsafe states (line 4). In the loop, the pre-images of (formulae in) P with respect to each transition in *Tr* are computed and simplified (line 8). Each pre-image is tested for fix-point as follows. The formula

$$\exists \underline{i}. \forall \underline{j}. P(\underline{i}) \quad \wedge \bigwedge_{V \in \mathsf{AV} \cup \mathsf{TBV}} \neg V(\underline{j})$$

is checked for satisfiability (line 9); this is done by the module Minst which goes over the set AV∪TBV and incrementally invokes the module inst to handle the instantiation of each formula V (see the dashed box in Figure 1(a)). Each (instance of a) formula V in $\mathsf{AV} \cup \mathsf{TBV}$ is incrementally added and checked in the current Yices context. (This is similar to the *forward redundancy elimination* of resolution-style automated theorem provers.) If Minst concludes the unsatisfiability of the formula, then a fix-point has been reached (as $P \rightarrow \bigvee \mathsf{AV} \cup \mathsf{TBV}$ is valid) and the formula is discarded; otherwise, it is added to the set TBV of the already visited states (line 9). Then, it is tested whether the computed pre-image has a non-empty intersection with the set of initial states and, if the case, unsafety of the array-based system is reported (line 10). Once all the pre-images of a certain set of states have been computed, a new set of states is selected (and deleted from) the set TBV of states yet to be visited, if TBV is non-empty and the main loop is executed again with the new choice of P.

Since the set AV is handled as a queue, MCMT explores *on-the-fly* (i.e. while generating it) and *breadth-first* (according to the order in which the user listed the transitions) the 'state space graph' of the safety problem under consideration. The nodes of this graph are labelled by the primitive differentiated formulae representing sets of backward reachable states. MCMT (using the option `-r`) is capable of producing a list of the visited nodes, the formulae labelling them, and a graphical representation of the state space graph. The first two in LaTeX while the last one in the format supported by the graphical tool GraphViz.[7]

3.2 Main modules

To complete the high-level description of MCMT, we now give more details about the heuristics embodied in each module of the architecture in Figure 1(a).

Pre. Let a be the array state variables of an array-based system, $P(a) := \exists \underline{k}\, P(\underline{k}, a[\underline{k}])$, where P is a primitive differentiated conjunction of literals, and $\tau(a, a') := \exists \underline{i}\ (G(\underline{i}, a[\underline{i}]) \wedge a' = \lambda j. F(\underline{i}, a[\underline{i}], j, a[j]))$ be a transition corresponding to a guarded assignment, where G is a conjunction of literals and F is a case-defined function update (the use of the λ-abstraction can be avoided by using universal quantification as seen for the Illinois protocol). It is straightforward to show [12] that $Pre(\tau, P)$ is equivalent to the following formula:

$$\exists \underline{i}\, \exists \underline{k}. G(\underline{i}, a[\underline{i}]) \wedge P(\underline{k}, F(\underline{i}, a[\underline{i}], \underline{k}, a[\underline{k}])). \qquad (2)$$

So, in principle, the implementation of *Pre* simply amounts to (repeatedly) build up formula (2). However, there are two crucial problems in using formula (2) without any further manipulation. First, the number of existentially quantified variables in the formula has grown since $\exists\ \underline{k}$ is augmented with $\exists\ \underline{i}$. Indeed, this put an additional burden on the quan-

[7] `http://www.graphviz.org/`

tifier instantiation routine of the module inst. It would be desirable to find ways to limit the growing number of existentially quantified variables in the prefix of the pre-image or, even better, to ensure it remains constant. To this end, the techniques described in [12] have been implemented in the module classify which is described below. The second problem in directly using (2) is that there is no guarantee it to be in primitive differentiated form. To avoid this problem, the module Pre performs a case-analysis according to the case-defined function update F so as to compute several primitive differentiated formulae whose disjunction is logically equivalent to (2). In order to reduce the number of inconsistent updates (i.e. unsatisfiable formulae) so as to reduce the number of formulae to be visited (i.e. added to TBV), a satisfiability check is required. While Yices can be invoked to do this, for efficiency reasons, it is better to minimize the number of calls to the available SMT solver. To this end, the module classify besides deciding how it is possible to control the grow of the existential prefix for each transition, it also derives useful information which allows for computationally inexpensive satisfiability checks (see below for more on this). If all the cheap tests are passed, Yices must indeed be invoked.

classify. As explained above, the goal of this module is to reduce the number of existentially quantified variables $\exists \underline{i}$ that should be added to $\exists \underline{k}$ in the prefix of (2). The idea to do this consists of trying to identify as many as possible variables in $\underline{i}$ with those already present in $\underline{k}$. This is justified by the observation that it is often the case that naively adding the extra variables in $\underline{i}$ yields primitive differentiated formulae which are immediately discharged by the successive fix-point test (line 9 of Figure 1(b)); thereby wasting computational resources. Under suitable hypotheses on the theories over indexes and elements, it is possible to predict whether this will happen before entering the main loop implementing backward reachability analysis as it depends on the form of the guards and the case-defined function updates used to define transitions (see [12] for details). Concretely, this is done by generating certain proof obligations for Yices, whose unsatisfiability imply that some or all of the additional variables in $\underline{i}$ can be identified with those in $\underline{k}$. For example, in the case of the Illinois protocol, classify establishes that transition `t2` may avoid to introduce a new (existentially quantified) variable while for `t1`, this is not the case and a new variables should be considered. The module classify is capable of establishing the behavior of transition with one and two existentially quantified variables and indicates whether 0, 1, or 2 new variables must be introduced by the module Pre. We call this technique *static instance reduction*. In the pseudo-code of Figure 1(b), we indicate that a transition τ have been analyzed by classify and the resulting information is attached to the transition using the label ℓ. Recall that τ^ℓ also contains additional information to allow for the cheap satisfiability tests mentioned above and used by Pre but also by inst (see below). To illustrate this point, consider an array variable a whose elements are an enumerated data-type. Then, τ^ℓ stores the value of a after the execution of the transition (since we perform a backward analysis, this is the only relevant value). In this way, to check for unsatisfiability, it is sufficient to search in a primitive differentiated formula, two literals of the form $a[i] = c$ and $a[i] = d$, for some index i, where c and d are two distinct 'values' of the enumerated data-type. We call this form of dynamic instance reduction as *filtering modulo enumerated data-types* and we will see another use of it for formulae describing sets of states below.

inst. As already said, one of the main bottle-necks in MCMT is the generation of instances for the checks of fix-point and non-empty intersection with the initial set of states. While the module classify attempts to *statically* control the number of instances of a universally quantified formula by reducing its number of universally quantified variables, our experience showed that this is not enough and we need also some technique to *dynamically* reduce the number of such instances. To this end, the key observation is that there is a large amount of instances which are useless in checking for unsatisfiability. To understand the problem, recall that the tool is required to check the satisfiability of the following kind of formulae:

$$\varphi(\underline{i}) \quad \wedge \quad \bigwedge\{\forall \underline{j}.\neg\psi(\underline{j}) \mid \psi \in \mathsf{AV} \cup \mathsf{TBV}\}$$

where φ and the ψ's are primitive differentiated formulae. Roughly, the algorithm implemented in inst consists of asserting φ in the actual Yices context and then incrementally enumerating all the instances of the ψ's. Indeed, before considering a new instance, a satisfiability check is performed. Now, consider again the case where there is an array variable a whose elements belong to an enumerated data-type. Assume that φ contains a literal of the form $a[i_k] = c$ (for $i_k \in \underline{i}$) and a ψ contains a literal of the form $a[j_l] = d$ (for $j_l \in \underline{j}$) such that c and d are two distinct values of the enumerated data-type. Then, it may happen—during the enumeration of the instances of a ψ—that j_l is instantiated to i_k and this implies that the instances under consideration are trivially satisfiable. To see this, rewrite φ as $a[i_k] = c \wedge \varphi'$ and the instance of ψ as $a[i_k] = d \wedge \psi'$ so as to obtain (after simple logical manipulations) the formula

$$(a[i_k] = c \wedge \varphi') \quad \wedge \quad (a[i_k] = d \rightarrow \neg\psi'),$$

which is trivially satisfiable, if φ is so. The module inst filters out useless instances by using this observation. This is another facet of the filtering modulo enumerated data-types technique applied to state formulae instead, as before, to transitions: also the primitive differentiated formulae produced by Pre are decorated with additional information such as the values in the arrays which range over enumerated data-types. Despite its simplicity, this heuristics dramatically reduces the numbers of instances which are passed to the SMT solver to check the satisfiability of a single universally quantified formula. To appreciate the importance of this heuristics, it is sufficient to observe that, to realize a fix-point check for a newly visited set of states (line 9 of Figure 1(b)), the module Minst is required to iterate the application of inst for each formula in the set AV of already visited states. Observe also that AV grows at each iteration of the backward analysis loop. Fortunately, our experience shows that few iterations in Minst are often sufficient and that we have a higher probability for an earlier detection of unsatisfiability when considering first those formulae which have been added to AV more recently. We call this form of dynamic instance reduction as *chronological fix-point checking*.

To conclude the description of the module inst, we mention two further heuristics. First, in the generate-and-check algorithm implemented in inst, the (incremental) satisfiability check is done after considering each new instance. It is possible to change the frequency of this test by using the option `-p`. Second, another heuristic belonging to the class of the dynamic instance reduction can be activated by the option `-f`. It consists of fixing the instantiation of a certain number of the universally quantified formulae after visiting a given number of sets of states. This makes the system incomplete but it may give significant speed-ups.

select. When describing the main loop of the tool in Figure 1(b), we claimed that the symbolic state space of the problem is explored breadth-first according to the order in which the user listed the transitions. This is not precise because of the function select used to extract the next set of states to be visited (line 12 of Figure 1(b)). In fact, along the lines of the dynamic instance reduction technique, we try to visit as a large part of the state space as possible by using first those formulae with fewer variables; it is possible to disable this heuristics by using the option `-b`.

simplify. This module performs some simple manipulations of the primitive differentiated formulae computed by Pre such as eliminating duplicated literals, orienting (according to some total ground ordering) equalities, or perform some simple normalization on arithmetic terms. It would be interesting to have some simplification routines used inside Yices available through the API so as to reuse the wealth of well-engineered techniques to simplify constraints for our tool and avoid to waste time duplicating functionalities.

4. EXPERIMENTS

MCMT has been implemented using the API light of Yices and release 0.2.1 can be downloaded at the following address `http://homes.dsi.unimi.it/~ghilardi/mcmt`.

As benchmarks, we have derived three sets of benchmarks for MCMT from the safety problems in [3, 2]: the first is of mutual exclusion protocols (with 7 problems, Table 1), the second is of cache coherence protocols (with 4 problems, Table 2), and the last is obtained as the counting abstraction of the previous problems (Table 3). We used the theory of finite linear orders as T_I for the first set of problems and the theory of equality as T_I for the other two. The theory T_E for the first two sets is the combination of an enumerated datatype theory for the control locations with theories for the data manipulated by the processes while for the third sets is Linear Arithmetics (over the Integers). The termination of the backward reachability analysis for some of these benchmarks specifying broadcast protocols and some versions of the mutual exclusion protocols (e.g., bakery) is guaranteed by applying results from [9]. In particular, it is possible to show termination when T_E is the theory of an enumerated datatype (i.e. each process in the parametrised system is modelled by a finite state automaton updating variables whose domains consist only of finitely many values). However, according to the experience in [7], even when termination is not guaranteed, we have obtained it in practice for all our benchmarks.

Columns 2-5 of the Tables report the statistics (timings are in seconds and obtained on a Pentium Dual-Core 3.4 GHz with 2 Gb Sdram) of the implementation of the main loop in Figure 1(b) while columns 6-9 show the results for the following variant (available in the latest release of the tool only). We have replaced the instruction at line 12 with the following block of code:

12_1 **do**
12_2 $\quad P \leftarrow \mathsf{select}(\mathsf{TBV});$
12_3 **while** $\mathsf{TBV} \neq \emptyset \wedge \mathsf{Minst}(\mathsf{AV} \cup \mathsf{TBV}, \mathsf{getinds}(P), P) = \mathsf{unsat};$

This is equivalent to check for fix-point a newly computed set of backward reachable states before computing its pre-images: if the test is positive (when the corresponding proof obligation is unsatisfiable), we can not only avoid to compute its pre-images but also eliminate it (recall that select has the side-effect of deleting the selected formula from its input set) so that its instances are no more considered when checking for fix-point or non-empty intersection with the set of initial states. (This technique is similar to the *backward redundancy elimination* in resolution-style theorem provers.) When the test for fix-point fails, the loop is exited and the primitive differentiated formula P starts the main loop in Figure 1(b).

Table 1 shows the usefulness of forward redundancy elimination as the seize of the problem grows. For the other two tables, the situation is less clear. However, we remark that all the cache coherence protocols that we have considered so far (except the German) are quite small and a brute force search of the tiny search space (see the column '#nodes') is likely to be more successful. Interestingly, there is some gain in using forward redundancy elimination on the last problem in this set (a difficult version of the German protocol, which is well-known to be a significant benchmark for verification tools).

Although a comparative analysis is somewhat difficult in lack of a standard for the specifications of safety problems, we report that MCMT performs comparably with the model checkers PFSand UNDIP on small to medium sized problems and outperforms them on larger instances.[8] These tools use a partially declarative approach by combining *ad hoc* algorithms to explore the symbolic state space of a system and constraint solvers for numerical data in order to reason on the elements manipulated by the processes. By comparing the statics of such systems with those of MCMT, it is apparent that a significantly smaller number of sets of states is visited during backward analysis. This is so because a single primitive differentiated formula is capable of representing a large amount of (minimal) configurations that a (partially) symbolic model checker, like PFS or UNDIP, is forced to enumerate. However, we must point that the completely symbolic representation of sets of states contains a lot of redundancy so that powerful heuristics are required to make our approach to scale up to large and interesting problems. The techniques for instance reductions described above can be seen as an important class of heuristics capable of handling this redundancy.

[8] `http://www.it.uu.se/research/docs/fm/apv/tools/{pfs,undip}`

	depth	#nodes	#calls	time	depth	#nodes	#calls	time
Bakery	9	29	221	**0.104**	9	28	252	0.124
Burns	14	57	497	0.216	14	57	429	**0.188**
Java M-lock	9	23	353	0.156	9	23	343	**0.152**
Dijkstra	13	40	392	0.148	13	38	256	**0.096**
Dijkstra (rv)	14	138	6905	5.756	14	127	4451	**3.324**
Szymanski	17	143	3266	2.208	17	136	3164	**2.036**
Szymanski (a)	23	2358	902017	24m19s	23	1745	475505	**11m19s**

Table 1: Mutual exclusion algorithms

	depth	#nodes	#calls	time	depth	#nodes	#calls	time
Mesi	3	2	175	0.032	3	2	177	**0.028**
Moesi	3	2	304	0.048	3	2	306	0.048
Illinois	4	8	998	**0.196**	4	8	1006	0.216
German	26	2985	322335	8m39s	26	2442	125060	**3m51s**

Table 2: Cache coherence protocols

	depth	#nodes	#calls	time	depth	#nodes	#calls	time
Mesi	3	2	18	**0.008**	3	2	22	0.012
Moesi	3	2	21	**0.016**	3	2	25	0.024
Illinois	3	4	82	0.036	3	3	56	**0.020**
German	9	13	46	0.028	9	13	63	0.028

Table 3: Cache coherence protocols: counting abstraction

Experimental results for a powerful heuristics for invariant generation on the same set of benchmarks are also reported in [11]. The command line option to active such a heuristics is `-i`.

5. DISCUSSION

We have described the architecture and the main loop of our model-checker MCMT based on a careful combination of SMT solving (Yices), a backward reachability algorithm, and heuristics to solve the proof obligations encoding the fix-point tests and the checks for non-empty intersection with the initial set of states.

We plan to extend MCMT in two directions. First, we intend to implement or to import a quantifier elimination module for the the variables ranging over the elements of the array (for some theories T_E admitting quantifier elimination, like for instance Linear Arithmetic and its fragments). Once this module is available, existentially quantified variables for elements can be used in the guards of the transitions: this will enable MCMT to deal, for example, with some of the benchmarks in [3]. Second, we want to support two-dimensional arrays to model communication channels between processes: this extension is crucial for MCMT to be able to treat—in a natural way—concurrent processes communicating over channels (see, e.g., [3] for examples of the verification of this kind of distributed systems).

6. REFERENCES

[1] P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *Proc. of LICS*, pages 313–321, 1996.

[2] P. A. Abdulla, G. Delzanno, N. B. Henda, and A. Rezine. Regular model checking without transducers. In *TACAS*, volume 4424 of *LNCS*, pages 721–736, 2007.

[3] P. A. Abdulla, G. Delzanno, and A. Rezine. Parameterized verification of infinite-state processes with global conditions. In *CAV*, volume 4590 of *LNCS*, pages 145–157, 2007.

[4] P. A. Abdulla, N. B. Henda, G. Delzanno, and A. Rezine. Handling parameterized systems with non-atomic global conditions. In *Proc. of VMCAI*, volume 4905 of *LNCS*, pages 22–36, 2008.

[5] L. de Moura and N. Bjørner. Efficient e-matching for smt solvers. In *Proc. of CADE*, LNCS, 2007.

[6] D. Déharbe and S. Ranise. Satisfiability solving for software verification. *Int. Journal on STTT*, 2009. To appear.

[7] G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *Proc. of CAV*, number 1855 in LNCS, 2000.

[8] Y. Ge, C. Barrett, and C. Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In *Proc. of CADE-21*, LNCS, 2007.

[9] S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Towards SMT Model-Checking of Array-based Systems. In *Proc. of IJCAR*, LNCS, 2008.

[10] S. Ghilardi and S. Ranise. A Note on the Stopping Failures Models. 2009. Draft, available from MCMT distribution.

[11] S. Ghilardi and S. Ranise. Goal-Directed Invariant Synthesis in Model Checking Modulo Thoeries. In *Proc. of TABLEAUX 09*, LNCS, 2009. Full version available at `http://homes.dsi.unimi.it/~ghilardi/allegati/GhRa-RI325-09.pdf`.

[12] S. Ghilardi, S. Ranise, and T. Valsecchi. Light-Weight SMT-based Model-Checking. In *Proc. of AVOCS 07-08*, ENTCS, 2008.

[13] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

Graph-based Reduction of Program Verification Conditions

J.-F. Couchot
LIFC
University of Franche-Comté
F-25030 Besançon
couchot@lifc.univ-fcomte.fr

A. Giorgetti
INRIA CASSIS / LIFC
University of Franche-Comté
F-25030 Besançon
alain.giorgetti@univ-fcomte.fr

N. Stouls
AMAZONES INRIA / CITI
INSA Lyon
F-69621 Villeurbanne
Nicolas.Stouls@insa-lyon.fr

ABSTRACT
Increasing the automaticity of proofs in deductive verification of C programs is a challenging task. When applied to industrial C programs known heuristics to generate simpler verification conditions are not efficient enough. This is mainly due to their size and a high number of irrelevant hypotheses.

This work presents a strategy to reduce program verification conditions by selecting their relevant hypotheses. The relevance of a hypothesis is determined by the combination of a syntactic analysis and two graph traversals. The first graph is labeled by constants and the second one by the predicates in the axioms. The approach is applied on a benchmark arising in industrial program verification.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms
Verification, Experimentation

Keywords
Proof, hypothesis selection

1. INTRODUCTION
Deductive software verification aims at verifying program properties with the help of theorem provers. It has gained more interest with the increased use of software embedded in, for instance, airplanes commands, cars or smart cards, requiring a high-level of confidence.

In the Hoare logic framework, program properties are expressed by first-order logical assertions on program variables (preconditions, postconditions, invariants, ...). The deductive verification method consists in transforming a program, annotated with sufficiently many assertions, into so-called *verification conditions* (VCs) that, when proved, establish that the program satisfies its assertions. In the KeY system [2] a special purpose logic and calculus are used to prove these verification conditions. The drawback of this approach is has it is specific to a programming language and a target prover. In contrast, a multi-prover approach is followed by effective tools such as ESC/Java [10] for Java programs annotated using the Java Modeling Language [4], Boogie [1] for the C# programming language, and Caduceus/Why [12] for C programs. The latter also offers Java as input programming language.

A theorem prover is invoked to establish the validity of each verification condition. One of the challenges in deductive software verification is to automatically discharge as many verification conditions as possible. A key issue is that the whole context of a verification condition is a huge set of axioms modelling not only the property and the program under verification, but also many features of the programming language. Simply passing this large context to an automated prover induces a combinatorial explosion, preventing the prover from terminating in reasonable time.

Possible solutions to reduce the VC size and complexity are to optimize the memory model (e.g. by introducing separations of zones of pointers [16]), to improve the weakest precondition calculus [17] and to apply strategies for simplifying VCs [14, 8, 18]. This work focuses on the latter. We suggest heuristics to select axioms to feed automated theorem provers (ATPs). Instead of blindly invoking ATPs with a large VC, we present reduction strategies that significantly prune their search space. The idea behind these strategies is quite natural: an axiom is relevant if a prover applies it successfully, i.e. without diverging, to establish the conclusion. Relevance criteria are computed by the combined traversal of two graphs representing symbol dependencies within axioms. In the graph of constants edges represent the conjoint presence of two constants in some ground axiom. In the graph of predicates arcs represent logical dependencies between predicates occuring in the same axiom.

In former work [5], selection was limited to ground hypotheses and comparison predicates were not taken into account. This led to unsatisfactory results, for instance when the conclusion is some equality between terms. The present work extends selection to context axioms, comparison predicates and hypotheses with quantifiers. We propose new heuristics that increase the number of automatically discharged VCs.

The plan of the article is as follows. Section 2 presents the industrial C example that has motivated this work. This case study is a part of the Oslo [3] secure bootloader annotated with a safety property. Section 3 presents the general structure of a verification condition. Section 4 shows how dependencies are stored in graphs. The selection strategy of hypotheses is presented in Section 5. These last two sections are the first contribution. The second contribution is the implementation of this strategy as a module of Caduceus/Why [12]. Section 6 presents experimentation results. Section 7 discusses related work, concludes and presents future work.

2. TRUSTED PLATFORM CASE STUDY

Some new challenges for axiom filtering are posed by the context of the PFC project on Trusted Computing (TC). PFC (meaning trusted platforms in French) is one of the SYSTEM@TIC Paris Region French cluster projects. The main idea of the TC approach is to gain some confidence about the execution context of a program. This confidence is obtained by construction, by using a *trusted chain*. A trusted chain is a chain of executions where each launched program is previously registered with a tamperproof component, such as the *Trusted Platform Module* (TPM) hardware chipset. In this context of TC, we focus on the Oslo [3] secure loader. This program is the first step of a trusted chain. It uses some hardware functionalities of recent CPUs (AMD-V or Intel-TET technologies) to initialize the chain and to launch the first program of the chain.

The main trusted chain properties are temporal, but some recent works [13, 15] propose a method to translate a temporal property into first-order logic annotations in the code. This method is systematic and generates a large amount of VCs, including quantifications and arrays with many links between them. Therefore, this approach is a good generator for VCs with a medium or low level of automaticity. Table 1 gives some factual information about the studied part of Oslo. The VCs of this benchmark are publicly available [24].

Oslo program and specification	
Code	≈ *1500 lines*
Specification	≈ *1500 lines (functional)*
Number of VCs	≈ *7300 VCs*
Observed part of Oslo	
Observed code	= *218 lines*
Specification	≈ *1400 lines (functional and generated)*
Number of VCs	= *771 VCs*

Table 1: Some Metrics about the Oslo Program

3. VERIFICATION CONDITIONS

The verification conditions (VC) we consider are first order formulae whose validity implies that a piece of annotated source code satisfies some property. This section describes the general structure of VCs generated by Caduceus/Why. A VC is composed of a *context* and a *goal*. This structure is illustrated in Fig. 1.

The context depends on the programming language. It is a first-order axiomatization of the language features used in the program under verification. Typical features are data

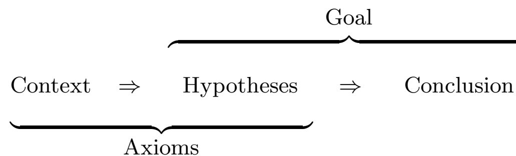


Figure 1: Structure of verification conditions

types or a memory model, enriched to allow the specification of, e.g. separated pointer regions. For instance, a typical VC produced by Caduceus/Why has a context with more than 80 axioms.

VCs are generated in the input format of many first-order ATPs, among which Simplify [9] and SMT solvers [6]. The Simplify automatic prover has a specific input language. SMT solvers such as Alt-Ergo and Yices have a common input language. Alt-Ergo is however addressed in the Why input language for more efficiency. For SMT solvers, the context is presented as a base theory, usually a combination of equality with uninterpreted function symbols and linear arithmetic, extended with a large set of specific axioms.

The goal depends on the program and on the property under verification. When this property is an assertion about a given program control point, the goal is generated by the weakest precondition (wp) calculus of Dijkstra [11] at that control point. The goal is considered as a *conclusion* implied by *hypotheses* that encode the program execution up to the control point.

Running example. Consider the following function:

```
struct p {
 int x;
} p;

struct t {
 struct p v[2];
} t;

/*@ requires \valid(a) &&
  @  (\forall int i; 0<=i<=1 => \valid(a->v[i]))
  @ assigns a->v[0].x */
void f(struct t *a) {
 a->v[0].x = 2;
}
```

The `requires` annotation specifies a precondition and the `assigns` annotation means that function `f` modifies no other location than `a->v[0].x`. The hypotheses of the generated VC are

$\mathsf{valid}(a)$,
$(\forall i : \mathsf{int}\,.\, 0 \leq i \leq 1 \Rightarrow \mathsf{valid}(a, \mathit{shift}(\mathit{acc}(m_v, a), i)) \wedge$
$\quad \mathsf{valid_acc}(m_{pPM}))$,
$\mathsf{valid_acc_range}(m_v, 2)$,
$\mathsf{separation1_range}(m_v, 2), \mathsf{valid_acc}(m_v)$,
$r = \mathit{acc}(m_v, a), r_0 = \mathit{shift}(r, 0)$, and $m_x_0 = \mathit{upd}(m_x, r_0, 2)$.

The conclusion is

$$\mathsf{not_assigns}(m_x, m_x_0, \mathit{singleton}(\mathit{acc}(m_v, a))).$$

The meaning of these formulae is as follows. m_{pPM} is the pointer (P) memory (M) for the structures of type `p`. valid_acc(m) means that the memory m is initialized, i.e. that this memory is accessible from any valid pointer in the allocation table. The first two hypotheses correspond to the precondition. In the next two hypotheses the predicates valid_acc_range(m_v, 2) and separation1_range(m_v, 2) respectively mean that any access to the memory m_v returns an array t such that pointers $t[0]$ and $t[1]$ are valid and $t[0] \neq t[1]$. The last three hypotheses come from a flattening-like decomposition of the statement `a->v[0].x = 2` performed by the VC generator. The function $\mathit{shift}(t, i)$ allows access to the index i in the array t. The conclusion translates the `assigns` annotation into a relation between two memory values. m_x is the value of memory x before execution of `f` and m_x_0 is its value after execution of `f`. The third parameter is the representation of `a->v[0]`. Our preprocessor eliminates the last three hypotheses and the intermediary constants that they introduce by considering that the conclusion is

$$\mathsf{not_assigns}(m_x, \mathit{upd}(m_x, \mathit{shift}(\mathit{acc}(m_v, a), 0), 2), \mathit{singleton}(\mathit{acc}(m_v, a))). \quad (C)$$

4. GRAPH-BASED DEPENDENCY

Basically, a conclusion is a propositional combination of potentially quantified predicates over some terms. Dependencies between axioms and the conclusion can then arise from terms and predicates. Terms in the goal may either come from the annotated program (from statements or assertions) or may result from a weakest precondition calculus applied to the program and its assertions. The term dependency just transcribes that parts of the goal (in particular, hypotheses and conclusion) share common terms. It is presented in Section 4.1. Two predicates are dependent if there is a deductive path leading from one to the other. The predicate dependency is presented in Section 4.2. Finally, Section 4.3 presents a special dependency analysis for comparison predicates.

4.1 Term Dependency

In order to describe how hypotheses connect terms together and according to previous work [5], an undirected connected graph G_c is constructed by syntactic analysis of term occurrences in each hypothesis of a VC. The graph vertices are labeled with the constants occurring in the goal and with new constants resulting from the following flattening-like process. A fresh constant f_i where i is some unique integer is created for each term $f(t_1, \ldots, t_n)$ in the goal. There is a graph edge between the two vertices labeled with the constants f_i and c when c is t_j if t_j is a constant and when c is the fresh constant created for t_j if t_j is a compound term ($1 \leq j \leq n$).

Running example. An excerpt of the graph representing the VC presented in Section 3 is given in Fig. 2. The vertices *shift*_6 and *acc*_7 come from the second hypothesis and the other vertices come from the conclusion (C).

4.2 Predicate Dependency

A weighted directed graph is constructed to represent implication relations between predicates in an efficient way.

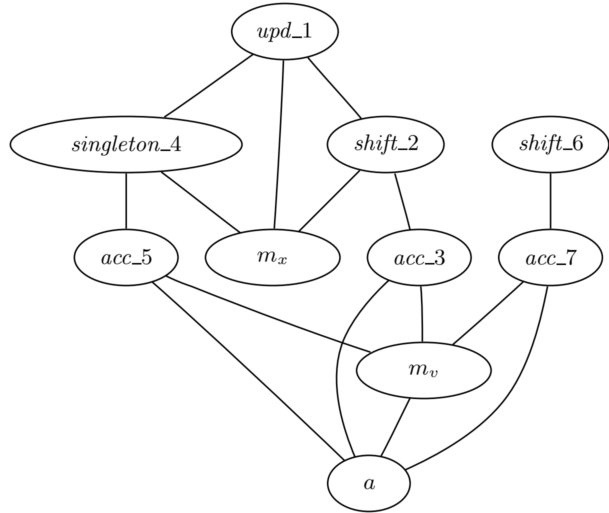


Figure 2: Example of Constant Dependency Graph

Intuitively, each graph vertex represents a predicate name and an arc from a vertex p to a vertex q means that p may imply q. What follows are details on how to compute this graph of predicates, named G_P. This section describes the general approach. The next section adds a special treatment for comparison predicates.

First, each context axiom is decomposed into a conjunctive normal form (CNF). It is done in a straightforward way (in contrast to optimised CNF decomposition [19]): axioms are of short size and their transformation into CNF does not yield a combinatorial explosion. The resulting clauses are called *axiom clauses*. Each graph vertex is labeled with a predicate symbol that appears in at least one literal of the context. If a predicate p appears negated (as $\neg p$) in an axiom clause, it is represented by a vertex labeled with $\overline{p}$. A clause is considered as a set of literals. For each axiom clause Cl and each pair $(l, l') \in Cl \times Cl$ of distinct literals in this clause, there is an arc in G_P depending on the polarity of l and l'. There are three distinct cases modulo symmetry to consider. They are enumerated in Table 2, where p and q are two distinct predicates. To reduce the graph size, the contraposite of each implication is not represented as an arc in the graph but is considered when traversing it, as detailed in Section 5.2.

Value of the (l, l') pair	Arcs
$(\neg p, q)$	$\{p \longrightarrow q\}$
(p, q)	$\{\overline{p} \longrightarrow q\}$
$(\neg p, \neg q)$	$\{p \longrightarrow \overline{q}\}$

Table 2: Translating Pairs of Literals into Arcs.

The intended meaning of an arc weight is that the lower the weight is, the higher the probability to establish q from p is. Therefore, the arc introduced for the pair (p, q) along Table 2 is labeled with the number of predicates minus one ($\mathit{card}(Cl) - 1$) in the clause Cl under consideration. For

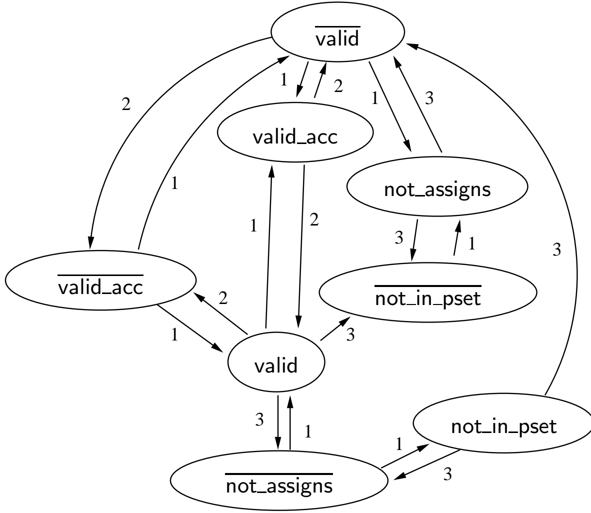


Figure 3: Example of Predicate Dependency Graph

instance, a large clause with many negative literals, with $\neg p$ among them, and with many consequents, with q among them, is less useful for a deduction step leading to q than the smaller clause $\{\neg p, q\}$. Finally, two weighted arcs $p \xrightarrow{w_1} q$ and $p \xrightarrow{w_2} q$ are replaced with the weighted arc $p \xrightarrow{min(w_1,w_2)} q$.

Running example. Figure 3 represents the dependency graph corresponding to the definition of predicates valid, not_assigns and valid_acc. It is an excerpt of the graph representing the memory model of Caduceus/Why.

4.3 Handling Comparison Predicates

In a former work [5], equalities and inequalities were ignored when memorizing predicate dependencies. This leads to unsatisfactory results when (in)equality is central for deduction, e.g. when the conclusion is some equality between terms. If we handle equality as the other predicates, the process of Section 4.2 connects too many vertices with the vertex labeled =. We have experienced that this reduction of the graph diameter has a negative impact on the quality of selection.

More generally the present section suggests a special construction of graph vertices and edges for comparison predicates. A comparison predicate is an equality =, an inequality $\neq$, a (reflexive) order relation ($\leq$ or $\geq$) or an irreflexive pre-order ($>$ or $<$). The keys of this construction are the support of types and the exploitation of some causalities between comparison predicates.

4.3.1 Typed comparisons

Each comparison predicate $\circ$ is written $\circ_t$ where $\circ$ is $=, \neq, \leq, <, \geq$ or $>$ and t is the type of the $\circ$ operands. For simplicity, the focus is on the types t where $\leq_t$ and $\geq_t$ are total orders, $>_t$ and $<_t$ are their respective reverse orders, and $\leq_t$ is the union of $<_t$ and $=_t$. A typical example is the type int of integers.

$$
\begin{aligned}
x \leq y \wedge y \leq x &\Rightarrow x = y && (1)\\
x = y &\Rightarrow x \geq y && (2)\\
x = y &\Rightarrow y \geq x && (3)\\
x > y &\Rightarrow x \geq y && (4)\\
x \geq y &\Rightarrow x > y \vee x = y && (5)
\end{aligned}
$$

Figure 4: Some Axioms Relating Comparison Predicates

Each comparison $t_1 \circ_t t_2$ present in at least one axiom is represented by two nodes respectively labeled with $\circ_t$ and $\overline{\circ_t}$, where $\overline{=_t}$, $\overline{\neq_t}$, $\overline{\leq_t}$, $\overline{<_t}$, $\overline{\geq_t}$, and $\overline{>_t}$ respectively are $\neq_t$, $=_t$, $>_t$, $\geq_t$, $<_t$, and $\leq_t$. For instance, the two nodes $\leq_{\mathsf{int}}$ and $>_{\mathsf{int}}$ represent a total order on integers and its negation. These labels are called the *typed* comparison predicates.

Apart from this difference in the definition of $\overline{\circ_t}$, the arcs connected to typed comparison predicates are constructed following the general rules described in Table 2.

4.3.2 Causalities between comparison predicates

Verification conditions are expressed as SMT problems in AUFLIA logics [22]. Since the comparison predicates between integers are interpreted in AUFLIA, no context axiom contributes to their definition. Figure 4 suggests such a list of axioms. To lighten the figure, the predicates are not indexed with int.

Adding these axioms to the context would be counterproductive. We propose instead to analyze them to enrich the predicate graph *as if* they were in the context. Since the algorithm of axiom selection does not take loops into account, the sole arcs of interest in the predicate graph are between distinct nodes. It is then impossible to proceed so on internal properties like reflexivity, irreflexivity, symmetry or transitivity. This is the reason why Figure 4 is limited to axioms between distinct predicates. The symmetric axioms where $\leq$ and $<$ respectively replace $\geq$ and $>$ are also treated but are not reproduced. The arcs resulting from the application of the rules of Table 2 to those ten axioms are added to the graph of predicates.

5. AXIOM SELECTION

Relevant axioms remain to be selected. Intuitively, an axiom is relevant with respect to a conclusion if a proof that needs this axiom can be found. Variables and predicates included in a relevant axiom are also called relevant.

Section 5.1 shows how to select relevant constants in, Section 5.2 how to select relevant predicates and Section 5.3 how to combine these results to select relevant axioms. A selection strategy is presented as an algorithm in Section 5.4.

5.1 Relevant Constants

A node in the graph of constants G_c is identified with its labeling constant. Let n be the diameter of the graph of constants G_c. Starting from the set $\mathcal{C}_0$ of constants in the conclusion, a breadth-first search algorithm computes the sets $\mathcal{C}_i$ of constants in G_c that are reachable from $\mathcal{C}_0$ with at

most i steps ($1 \leq i \leq n$). Finally, unreachable constants are added to the limit of the sequence $(\mathcal{C}_n)_{n\in\mathbb{N}}$ for completeness. Let $\mathcal{C}_\infty$ be the resulting set.

To introduce more granularity in the computation of reachable constants, we propose as a heuristic to insert nodes that are linked several times before nodes that are just linked once. Semantically it gives priority to constants which are closer to the conclusion. Notice that, in this case, the index i of $\mathcal{C}_i$ does not correspond to a path length anymore.

Running example. The sequence of reachable constants sets associated to the graph in Fig. 2 is:

$$\begin{array}{lll} \mathcal{C}_0 & = & \{m_x, m_v, a\}, \\ \mathcal{C}_1 & = & \mathcal{C}_0 \cup \{acc_3, acc_5, acc_7\}, \\ \mathcal{C}_2 & = & \mathcal{C}_1 \cup \{singleton_4, shift_2\}, \\ \mathcal{C}_3 & = & \mathcal{C}_2 \cup \{shift_6\}, \\ \mathcal{C}_4 & = & \mathcal{C}_3 \cup \{upd_1\} \text{ and} \\ \mathcal{C}_\infty & = & \mathcal{C}_4. \end{array}$$

5.2 Relevant Predicates

A predicate p is identified with the vertex labeled p and its negation with the vertex labeled $\overline{p}$ in the graph of predicates G_P. A predicate symbol p is relevant w.r.t. a predicate symbol q if there is a path from p to q in G_P, or dually from $\overline{q}$ to $\overline{p}$. Intuitively, the weaker the path weight is, the higher the probability of p to establish q is. Relevant predicates extracted from G_P are stored into an increasing sequence $(\mathcal{L}_n)_{n\in\mathbb{N}}$ of sets. The natural number n is the maximal weight of paths considered in the graph of predicates.

We now present how $\mathcal{L}_n$ is computed. The conclusion is assumed to be a single clause. $\mathcal{L}_0$ gathers the predicates from the conclusion. For each predicate symbol p that is not in $\mathcal{L}_0$, a graph traversal computes the paths with the minimal weight w from p to some predicate in $\mathcal{L}_0$.

Furthermore, contraposition of each implication is considered: let p_1 and p_2 be two node labels, corresponding either to a positive or a negative literal. If the arc $p_1 \xrightarrow{w} p_2$ is taken into account, its couterpart $\overline{p_2} \longrightarrow \overline{p_1}$ is too, with the convention that $\overline{\overline{p}}$ is p. Let n be the minimal distance from $\mathcal{L}_0$ to the deepest reachable predicate. For $1 \leq i \leq n$, $\mathcal{L}_i$ is the set of vertices of G_P whose distance to $\mathcal{L}_0$ is less than or equal to i. $\mathcal{L}_\infty$ is the limit $\bigcup_{i\geq 0} \mathcal{L}_i$ augmented with the vertices from which $\mathcal{L}_0$ is not reachable.

Running example. From the predicate graph of the running example, depicted in Fig. 3 without the comparison predicates for lack of space, the first five sets of reachable predicates are

$$\begin{array}{lll} \mathcal{L}_0 & = & \{\mathsf{not_assigns}\}, \\ \mathcal{L}_1 & = & \mathcal{L}_0 \cup \{\overline{\mathsf{valid}}, \overline{\mathsf{not_in_pset}}, =\}, \\ \mathcal{L}_2 & = & \mathcal{L}_1 \cup \{\overline{<_{\mathsf{int}}}, \overline{\mathsf{valid_acc}}, \overline{\leq_{\mathsf{int}}}\}, \\ \mathcal{L}_3 & = & \mathcal{L}_2 \cup \{\mathsf{valid_acc}, \overline{>_{\mathsf{int}}}, \overline{\neq_{\mathsf{int}}}, \overline{\geq_{\mathsf{int}}}\} \text{ and} \\ \mathcal{L}_4 & = & \mathcal{L}_3 \cup \{\overline{=}, \mathsf{not_in_pset}, \mathsf{valid}, \leq_{\mathsf{int}}, \overline{=_{\mathsf{int}}}\}. \end{array}$$

5.3 Selection of Relevant Axioms

In this section, we present the main principles of the axiom selection combining predicate and constant selection. A first part describes hypothesis selection and a second one extends the approach to axioms from the context.

Let $(\mathcal{L}_n)_{n\in\mathbb{N}}$ and $(\mathcal{C}_n)_{n\in\mathbb{N}}$ respectively be the sequences of relevant predicate and constant sets. Let i be a counter which represents the depth of predicate selection. Similarly, let j be a counter corresponding to the depth of constant selection.

5.3.1 Hypothesis Selection

Let Cl be a clause from a hypothesis. Let V be the set of constants of Cl augmented with constants resulting from flattening (see Section 4.1). Let P be the set of predicates of Cl. The clause Cl should be selected if it includes constants or predicates that are relevant according to the conclusion. Different criteria can be used to verify this according to its sets P and V. Possible choices are, in increasing order of selectivity

1. the clause includes at least one relevant constant or one relevant predicate:
$$V \cap \mathcal{C}_j \neq \emptyset \ \vee \ P \cap \mathcal{L}_i \neq \emptyset$$
2. the clause includes more than a threshold t_v of relevant constants or more than a threshold t_p of relevant predicates:
$$card(V \cap \mathcal{C}_j)/card(\mathcal{C}_j) \geq t_v \ \vee card(P \cap \mathcal{L}_i)/card(\mathcal{L}_i) \geq t_p$$
3. all the clause constants and clause predicates are relevant:
$$V \subseteq \mathcal{C}_j \ \wedge \ P \subseteq \mathcal{L}_i$$

Our experiments on these criteria have shown that a too weak criterion does not accomplish what it is designed for: too many clauses are selected for few iterations, making the prover quickly diverge. Thus, we only consider the strongest criterion (3).

We have also often observed the case where only a conjunctive part of a universally quantified hypothesis is relevant. In that case, we split the conjunctive hypothesis into its parts and the filtering criterion is applied to the resulting predicates. A particular case is considered if a whole splittable hypothesis is relevant according to the criterion. Indeed, we then consider the original formula, in order to preserve its structure, which can be exploited by provers.

5.3.2 Context Axioms

Consider now the case of selecting relevant axioms from the context. Intuitively, an axiom of the context has to be selected if one of the predicate relations it defines is relevant for one hypothesis, i.e. the corresponding arc is used in the computation of $\mathcal{L}_i$. Practically, for each arc that is passed through while generating $\mathcal{L}_i$, we keep all the axioms of the context that have generated this arc.

5.4 Selection Strategy

The selection strategy experimented in this work is described in Fig. 5. The algorithm takes three parameters in input:

- a *VC* whose satisfiability has to be checked,
- a satisfiability solver *Prover*, and

```
Parameters : VC, Prover, TO
  // Prover call without VC reduction
  Res := Prover(VC,TO)
  if Res = timeout then
    i_max := 1 + Min depth giving reachable preds (VC)
    j_max := 1 + Min depth giving reachable vars (VC)

    i := 0;
    j := 0;
    While Res ≠ unsat ∧ i ≤ i_max do
      // Prover call after VC reduction
      Res := Prover(selection(VC, i, j), TO)
      j := j + 1;
      if j > j_max then
        i := i + 1;
        j := 0;
  return Res;
```

Figure 5: General Algorithm Discharging a VC with Axiom Selection

- a maximal amount of time TO given by the user to the satisfiability solver to discharge the VC.

The algorithm starts with a first attempt to discharge the VC without axiom selection. It stops if this first result is unsatisfiable or satisfiable. Notice that in the latter case, removing axioms cannot modify the result. Otherwise, *Prover* is called following an incremental constant-first selection.

The two natural numbers i_{max} and j_{max} are depth bounds for $\mathcal{L}_i$ and $\mathcal{C}_j$ computed during predicate graph and constant graph traversals. Since we want to reach $\mathcal{L}_\infty$ and $\mathcal{C}_\infty$, i_{max} and j_{max} are initially computed by the tool as one plus the minimal depth to obtain all reachable predicates and constants. This is interpreted by the tool as the ∞ depth, according to Sec. 5.2 and 5.1 (all predicates and constants of the graphs).

The *selection* function implements the selection of axioms (from context or hypotheses) according to the strongest criterion (3). Discharging the resulting reduced VC into a prover can yield three outcomes: satisfiable, unsatisfiable or timeout.

1. If the formula is declared to be unsatisfiable, the procedure ends. Adding more axioms cannot make the problem satisfiable.

2. If the formula is declared to be satisfiable, we may have omitted some axioms; we are then left to increment either i or j, i.e. to enlarge either the set of selected predicates or the set of selected constants.

   However, allowing predicates has a more critical impact than allowing new constants, since constants do not appear in context axioms. Therefore we recommend to first increment j, increasing $\mathcal{C}_j$ until eventually $\mathcal{C}_\infty$, before considering incrementing i. In this later case, j resets to 0.

3. If the formula is not discharged in less than a given time, after having iteratively incremented i and j, then the algorithm terminates.

6. EXPERIMENTS

The proposed approach is included in a global context of annotated C program certification. A separation analysis that strongly simplifies the verification conditions generated by a weakest precondition calculus, and thus greatly helps to prove programs with pointers has been proposed by T. Hubert and C. Marché [16]. Their approach is supported by the Why tool. The pruning heuristics presented here are developed as a post-process of this tool.

Section 6.1 gives some implementation and experimentation details. Section 6.2 presents experimental results on an industrial case study for trusted computing. This case study raises new challenges associated to the certification of C programs annotated with a temporal logic formula. Section 6.3 finally gives results obtained on a public benchmark.

6.1 Methodology

All the strategies presented in this work are implemented in OCaml as modules in the Why [12] tool in less than 1700 lines of code. Since these criteria are heuristics, their use is optional, and Why has command line arguments which allow a user to enable or disable their use. In the current version, several others heuristics have been developed, which are not considered because their impact on the performance of Why seems to be less obvious. In order to use the presented algorithms, the arguments to include in the Why call are:

```
--prune-with-comp --prune-context --prune-coarse-pred-comp
              --prune-vars-filter CNF
```

The first parameter includes comparison predicates in the predicate dependency graph. The second one requires filtering not only hypotheses but also axioms from the context. The third one requires to ignore arc weights. This option gives better execution times on the Oslo benchmark. Finally, the fourth argument requires for rewriting hypotheses into CNF before filtering.

The whole experiment is done on an Intel T8300@2.4GHz with 4Gb of memory, under a x86_64 Ubuntu Linux.

6.2 Results of Oslo Verification

First of all, among the 771 generated VCs, 741 are directly discharged, without any axiom selection. Next, the approach developed in [5] increases the result to 752 VCs.

Among the remaining unproved VCs, some rely on quantified hypotheses and others need comparison predicates that are not handled in the previous work [5]. They have motivated the present extensions, namely CNF reduction, comparison handling and context reduction. Thanks to these improvements, 10 more VCs are automatically proved by using the algorithm described in Fig. 5 with the three provers Simplify, Alt-Ergo 0.8 and Yices 1.0.20 with a timeout TO of 10 seconds.

The i_{max} and j_{max} limits depend on the VCs. Their observed values do not go beyond $i_{max} = 6$ and $j_{max} = 7$. These limits

express the number of versions in which the VCs have been cut. If edge weights are considered, then i_{max} grows up to $i_{max} = 18$ and the execution time is twice as long. Figure 6 sums up these results.

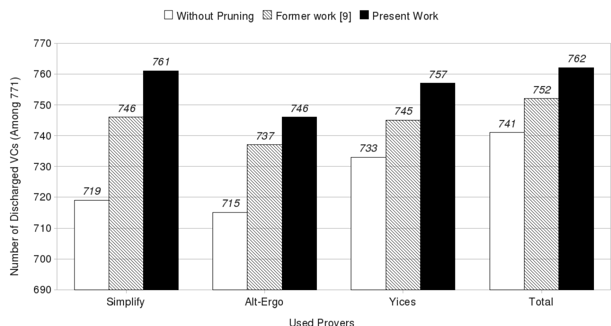


Figure 6: Result Comparison on Oslo Benchmark (771 VCs)

6.3 Public Why Benchmark

Our approach is developed in the Why tool, which translates Why syntax into the input syntax of several proof assistants (Coq, HOL 4, HOL Light, Isabelle/HOL, Mizar, PVS) and automated theorem provers (Alt-Ergo, CVC3, Simplify, Yices, Z3). This section shows some experimental results on the Why public benchmark[1].

The Why benchmark is a public collection of VCs generated by Caduceus or Krakatoa. These tools generate VCs respectively from C and Java programs, according to CSL and JML specifications. Hence, it partially matches to our requirements, since our work is focusing on the verification of VCs generated by these tools. The only limitation is that our method is focusing on VCs with a large amount of hypotheses, in contrast to the ones presented in this benchmark.

This benchmark is provided in two versions corresponding to two different pre-processes. Our results are similar with both versions. Alt-Ergo discharges 1260 VCs directly and 1297 VCs with axiom selection, adding 3 VCs to the 1310 VCs directly discharged by Simplify.

7. RELATED WORK AND CONCLUSION

We have presented a new strategy to select relevant hypotheses in formulae coming from program verification. To do so, we have combined two separate dependency analyses based on graph computation and graph traversal. Moreover, we have given some heuristics to analyse the graphs with a sufficient granularity. Finally we have shown the relevance of this approach with a benchmark issued from a real industrial code.

Strategies to simplify the prover's task have been widely studied since automated provers exist [28], mainly to propose more efficient deductive systems [28, 27, 26]. The KeY deductive system [2] is an extreme case. It is composed of a large list of special purpose rules dedicated to JML-annotated JavaCard programs. These rules make unnecessary an explicit axiomatization of data types, memory model, and program execution. Priorities between deduction rules help in effective reasoning. Beyond this, choosing rules in that framework requires as much effort as choosing axioms when targeting general purpose theorem provers.

[1] `http://proval.lri.fr/why-benchmarks/`

The present work can be compared with the set of support (sos) selection strategy [28, 20]. This approach starts with asking the user to provide an initial sos: it is classically the conclusion negation and a subset of hypotheses. It is then restricted to only apply inferences with at least one clause in the sos, consequences being added next into the sos. Our work can also be viewed as an automatic guess of the initial sos guided by the formula to prove. In this sense, it is close to [18] where initial relevant clauses are selected according to syntactical criteria, i.e. counting matching rates between symbols of any clause and symbols of clauses issued from the conclusion. By considering syntactical filtering on clauses issued from axioms and hypotheses, this latter work does not consider the relation between hypotheses, formalized by axioms of the theory: it provides a reduced forward proof. In contrast, by analyzing dependency graphs, we simulate natural deduction and are not far from backward proof search. By focusing on the predicative part of the verification condition, our objectives are dual to those developed in [14]: this work concerns boolean verification conditions with any boolean structure whereas we treat predicative formulae whose symbols are axiomatized in a quantified theory. Even in a large set of context axioms, most of the time, each verification condition only requires a tiny portion of this context. In [23, 7] a strategy to select relevant context axioms is presented, but it needs a preliminary manual task classifying axioms. Our predicate graph computation makes this axiom classification automatic. Recent advances have been made in the direction of semantic selection of axioms [25, 21]. Briefly speaking, at each iteration, the selection of each axiom depends on the fact whether a computed valuation is a model of the axiom or not. By comparison, our syntactical axiom selection is more efficient, indeed linear in the size of the input formula.

In a near future we plan to apply the strategy to other case studies. We also plan to investigate the impact on execution time of various strategies discharging the same list of verification conditions. We want to confirm or infirm with other benchmarks that weighting predicate dependencies with a formula length has no positive impact on automaticity but has a significant negative impact on the execution time. We also plan to integrate selection strategies in the Why tool or in a target automated theorem prover.

8. ACKNOWLEDGMENTS

This work is partially funded by the French Ministry of Research, thanks to the CAT (C Analysis Toolbox) RNTL (Reseau National des Technologies Logicielles), by the SYSTEM@TIC Paris Region French cluster, thanks to the PFC project (Plateforme de Confiance, trusted platforms), and by the INRIA, thanks to the CASSIS project and the CeProMi ARC. The authors also want to thank Christophe Ringeissen and the four anonymous referees for their insightful comments.

9. REFERENCES

[1] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.

[2] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[3] Bernhard Kauer. OSLO: Improving the security of Trusted Computing. In *16th USENIX Security Symposium, August 6-10, 2007, Boston, MA, USA*, 2007.

[4] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. Technical Report NIII-R0309, Dept. of Computer Science, University of Nijmegen, 2003.

[5] J.-F. Couchot and T. Hubert. A Graph-based Strategy for the Selection of Hypotheses. In *FTP 2007 - International Workshop on First-Order Theorem Proving*, Liverpool, UK, Sept. 2007.

[6] L. M. de Moura, B. Dutertre, and N. Shankar. A tutorial on satisfiability modulo theories. In W. Damm and H. Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 20–36. Springer, 2007.

[7] D. Deharbe and S. Ranise. Satisfiability Solving for Software Verification. Submitted in 2006. See `http://www.loria.fr/~ranise/pubs/sttt-submitted.pdf`.

[8] E. Denney, B. Fischer, and J. Schumann. An empirical evaluation of automated theorem provers in software certification. *International Journal on Artificial Intelligence Tools*, 15(1):81–108, 2006.

[9] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[10] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report 159, Compaq Systems Research Center, Dec. 1998. See also `http://research.compaq.com/SRC/esc/`.

[11] E. W. Dijkstra. *A discipline of programming*. Series in Automatic Computation. Prentice Hall Int., 1976.

[12] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.

[13] A. Giorgetti and J. Groslambert. JAG: JML Annotation Generation for Verifying Temporal Properties. In Luciano Baresi and Reiko Heckel, editors, *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006*, volume 3922 of *Lecture Notes in Computer Science*, pages 373–376. Springer, 2006.

[14] E. P. Gribomont. Simplification of boolean verification conditions. *Theoretical Computer Science*, 239(1):165–185, 2000.

[15] J. Groslambert and N. Stouls. Vérification de propriétés LTL sur des programmes C par génération d'annotations. In *AFADL'09*, 2009. Short paper.

[16] T. Hubert and C. Marché. Separation analysis for deductive verification. In *Heap Analysis and Verification (HAV'07)*, Braga, Portugal, Mar. 2007.

[17] K. R. M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, 2005.

[18] J. Meng and L. Paulson. Lightweight relevance filtering for machine-generated resolution problems. In *ESCoR: Empirically Successful Computerized Reasoning*, 2006.

[19] A. Nonnengart and C. Weidenbach. Computing Small Clause Normal Forms. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 6, pages 335–367. Elsevier Science, 2001.

[20] D. A. Plaisted and A. H. Yahya. A relevance restriction strategy for automated deduction. *Artificial Intelligence*, 144(1-2):59–93, 2003.

[21] P. Pudlak. Semantic selection of premisses for automated theorem proving. In G. Sutcliffe, J. Urban, and S. Schulz, editors, *CEUR Workshop Proceedings*, volume 257, pages 27–44, 2007.

[22] S. Ranise and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `http://www.SMT-LIB.org`, 2006.

[23] W. Reif and G. Schellhorn. Theorem proving in large theories. In M. P. Bonacina and U. Furbach, editors, *Int. Workshop on First-Order Theorem Proving, FTP'97*, pages 119–124. Johannes Kepler Universität, Linz (Austria), 1997.

[24] N. Stouls. Hypotheses selection applied to trusted computing. `http://perso.citi.insa-lyon.fr/nstouls/tools/oslo/`.

[25] G. Sutcliffe and Y. Puzis. Srass - a semantic relevance axiom selection system. In Springer, editor, *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, volume 4603 of *Lecture Notes in Computer Science*, pages 295–310, 2007.

[26] L. Wos. Conquering the meredith single axiom. *Journal of Automated Reasoning*, 27(2):175–199, 2001.

[27] L. Wos and G. W. Pieper. The hot list strategy. *Journal of Automated Reasoning*, 22(1):1–44, 1999.

[28] L. Wos, G. A. Robinson, and D. F. Carson. Efficiency and completeness of the set of support strategy in theorem proving. *J. ACM*, 12(4):536–541, 1965.