David Basin
Burkhart Wolff (Eds.)

Theorem Proving in Higher Order Logics

16th International Conference, TPHOLs 2003
Rome, Italy, September 2003
Emerging Trends Proceedings

Volume Editor

David Basin
ETH Zentrum
CH-8092 Zürich, Switzerland
E-mail: basin@inf.ethz.ch

Burkhart Wolff
Albert-Ludwigs-Universität Freiburg
D-79110 Freiburg, Germany
E-mail: wolff@informatik.uni-freiburg.de

Universität Freiburg
Technical Report No. 187

Preface

This volume constitutes the emerging trends proceedings of the *16th International Conference on Theorem Proving in Higher Order Logics* (TPHOLs 2003) held in Rome, Italy, September 8–12, 2003. TPHOLs covers all aspects of theorem proving in higher order logics as well as related topics in hardware verification and synthesis, verification of security and communication protocols, software verification, transformation and refinement, compiler construction and refinement.

TPHOLs 2003 follows the conference tradition of having both a *completed work* and a *work-in-progress* track. The papers of the first track were formally refereed, and published as volume 2758 of the Springer LNCS. This supplementary proceedings records work accepted in the work-in-progress category, and is intended to document emerging trends in higher order logic research and has been published as a technical report no. 187 of the University Freiburg. Papers in the work-in-progress track are reviewed prior to their acceptence. The work-in-progress track is regarded as an important feature of the conference as it provides a venue for the presentation of ongoing research projects, where researchers invite discussion of preliminary results.

This year, TPHOLs was co-located with *TABLEAUX*, the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, and with *Calculemus*, the Symposium on the Integration of Symbolic Computation and Mechanized Reasoning.

We would like to thank members of both the Freiburg and Zürich groups for their help in organizing the program. In particular, Achim D. Brucker, Barbara Geiser, and Paul Hankes Drielsma. We would also like to express our thanks to Marta Cialdea Mayer and her team for coordinating the local arrangements in Rome.

Finally, we thank our sponsors: Intel, ITT, ETH Zürich, and the Universität Freiburg. We also greatfully acknowledge the use of computing equipment from Universitá Roma III.

July 2003

David Basin, Burkhart Wolff
TPHOLs 2003 Program Chairs

Contents

Part I

Mathematical Theories

A Framework for Property Preservation Based on Galois Connections

Steffen Helke and Florian Kammüller

Technische Universität Berlin
Institut für Softwaretechnik und Theoretische Informatik

Abstract. We suggest to use the theorem prover Isabelle as a framework to support reasoning and construction processes involving property preservation. Galois connections are formalized in Isabelle/HOL on a very general level. We proof the equivalence with an abstract formulation of property preservation. Two possible application areas of the framework are briefly described: abstraction for statecharts and construction of refinements.

1 Introduction

A Galois connection is a pair of functions $\varepsilon : A \to B$ and $\pi : B \to A$ over partial orders $(A, \sqsubseteq)$ and $(B, \sqsubseteq)$. The functions ε and π correspond to each other in the sense that

$$\varepsilon\ x \sqsubseteq y\ \equiv x \sqsubseteq \pi\ y\,.$$

This simple algebraic concept is similar to that of an adjoint in category theory. It quite naturally corresponds to a notion of abstraction and is often used as a theoretical basis in work related to abstraction or refinement e.g. [SS99,L+95]. Its simplicity implies a certain lack of strong properties [BKS01,BKS00] but its conciseness makes it well suited as a theoretical frame in particular for computer science [MSS86,MMS02].

In this paper we present a concept for employing the simple but elegant theory of Galois connections as a framework in Isabelle [Pau94] for supporting reasoning processes that are needed in formal software engineering. Since they are algebraic they are elegant in the sense that their properties may be used in a calculational style to reason on an abstract level about concepts expressed as a Galois connection. However, as pointed out in [Chou96] when reasoning about property preservation, in the context of abstraction or refinement, Galois connection theory is usually a bit clumsy. In his paper Chou shows that the property preservation results presented by Loiseaux *et al.* in [L+95] are much simpler expressed in terms of relations. His notion of property preservation is based on the simple formula that a pair (p, q) of predicates on a concrete and an abstract level is a property preservation if

$$\forall (x, y) \in R.p\ x \Rightarrow q\ y\,.$$

Chou is right as far as one is concerned with showing general results about property preservation and abstraction. Once these results are established and we face the task of working with concrete applications the advantages of Galois connections as a more algebraic notion take over. Also Galois connections are functions and hence have better properties than relations when reasoning is concerned. Fortunately, as Chou already mentions, but does not exploit [Chou95], there is a unambiguous relationship between relational view and Galois connections.

In order to get the best out of both worlds we combine the two notions and suggest to use the combined framework to switch between the two representations as suits us best for the application at hand. We present a concept that is constituted by a formalization of the basis of Galois connections and predicate transformers showing the aforementioned correspondence with the relational view. The intention is to use this framework as a backbone for reasoning with refinement and abstraction.

One application is the formalization of statecharts in Isabelle [HK01]. Abstraction needs to be employed to reduce infinite state charts to finite ones. The resulting abstraction may then be passed on to a model checker via a specially constructed oracle interface.

A second planned application lies in constructing refinements from predicate translations [HKS02]. The idea is that system developers have a predicative view of system refinement. The framework shall be used to infer retrieve relations from predicate translations.

Galois connections have been formalized in Isabelle before [Gl01]. In this work Galois connections are expressed as a record type and the defining properties are given as predicates independently. This kind of formalization is well suited to reason about algebraic structures, but unfortunately did not suit our purpose of using the concept as a general framework. Also, in [Gl01] not many properties are proved about Galois connections, but instead they are integrated into a much bigger formalization of category theory. They serve there mainly as an application object rather than a concept that shall be used for other applications.

In this paper we first present our formalization of Galois connection in Section 2. Building on that Section 3 shows how the relational view can be derived from the Galois connections and how the approach relates to property preservation. Then we illustrate briefly how we intend to apply this framework to the abstraction process of statecharts and the construction of refinements in Section 4. Conclusions are given in Section 5.

2 Galois Connections

The theory of Galois connections has to be as general as possible in order to be applicable as a framework to other Isabelle formalizations. The most abstract way to formalize in Isabelle/HOL is given by axiomatic type classes, axclasses for short. Unfortunately, for known reasons, this concept is restricted to classes that only have one type parameter. This means that Galois connections cannot be expressed directly by axclasses, as they connect two different domains, that

is in general to arbitrary types. We would need exactly two type variables to be able to express the concept of Galois connections at the type class level.

The best we can do is to express Galois connections using a polymorphic type definition that demands that its type parameters are partial orders. To that end we can employ axclasses. Partial orders are an ideal application for axclasses.

Therefore, prior to the theory of Galois connections we introduce partial orders with top and bottom elements. We call them `pre_cl` as they may be seen as a primitive form of complete lattices. For a complete lattice we would need the stronger property that each subset has a least upper and greatest lower bound — not just the entire set [DP02] [1]. The class `pre_cl` is defined as an extension of the standard axclass `order` of partial orders in HOL. That way, we keep the current theory applicable to all future HOL developments.

```
axclass pre_cl < order
  Top_ax : "∃ T. ∀ x. x ⊑ T"
  Bot_ax: "∃ B. ∀ x. B ⊑ x"
```

We have just assumed the existence of some elements that may be named top and bottom. Their unique existence may be proved from the definitions. To be able to simply use the usual notion of Top and Bottom instead of dealing with an existentially quantified formula, we use the Hilbert choice operator `SOME` to introduce the following constants.

```
constdefs
  Top :: "α :: pre_cl"   "Top == SOME x. ∀ y. y ⊑ x"
  Bot :: "α :: pre_cl"   "Bot == SOME x. ∀ y. x ⊑ y"
```

However, we want to use the properties of top and bottom which is notoriously difficult with constants defined by Hilbert choice. Therefore we immediately infer them for future use.

```
∀ x. (x :: α :: pre_cl) ⊑ Top
∀. Bot ⊑ (x :: α :: pre_cl)
```

The definition of `pre_cl` is meaningful as is shown by the following `instance` section, i.e. the instantiation of the HOL set type constructor into the newly defined axclass.

```
instance set :: (type) pre_cl
```

This theorem shows that for any HOL 'type' the type constructor set produces a `pre_cl` type from it. This is obvious as sets always have top element `UNIV` and bottom element {} independent of the base type.

Now, we are prepared for the central type definition of a Galois connection.

```
typedef (α :: pre_cl, β :: pre_cl)GC =
     "{ (ε :: α => β, π :: β => α).
             (∀ x y. (ε x ⊑ y) = (x ⊑ π  y)) }"
```

[1] Complete lattices are defined in Isabelle/HOL in a similar style using axclasses by Wenzel. The concept of Galois connections is clearly also valid for complete lattices, but is more generally described our way.

This definition is in so far general as it defines Galois connections as a concept for any type that is an element of the class `pre_cl`. In order to define algebraic concepts properly one would normally define them using sets and dependent types to be able to reason about the structures. However, sometimes and so here, we just need an algebraic concept as a frame to enclose common features of other entities and are not interested in reasoning about the algebraic structure itself.

The generality guarantees that we may use Galois connections as a basis for other concepts expressed in Isabelle. The assumption that is implicit in the type definition is that a Galois connection may only exist over types that are at least elements of the class `pre_cl`. This is a prerequisite for a Galois connection as becomes most obvious when being forced to prove the non-emptiness of the defined type. The witness is

```
( λ  x :: α :: pre_cl. Bot :: β :: pre_cl, λ x . Top )
```

a pair of functions where the first element, i.e. ε, maps everything to bottom and the second element, i.e. π, maps everything to top. This is clearly a Galois connection for any HOL type that contains a top and a bottom element, because both sides of the defining equivalence are true.

```
ε G x ⊑ y ≡ Bot ⊑ y ≡ True ≡ x ⊑ Top ≡ x ⊑ π G y
```

Note that ε and π now denote the selector functions for the type `GC` producing the ε or π component of Galois connection `G`, respectively. From the type definition we can simply deduce that for any element of the new type the defining property holds

```
(ε G x ⊑ y) = (x ⊑ π G y)
```

which in turn implies the following useful equalities.

```
ε G (π G y) ⊑ y
x ⊑ π G( ε G x)
```

The latter may be used to infer that ε as well as π are monotonic functions. Further typical properties of Galois connections are:

```
ε G o π G o ε G = ε G
π G o ε G o π G = π G
```

We can hide the parameter `G` using locales in order to get syntax identical to mathematical notation.

3 Predicate Transformers and Property Preservation

After these preparations we are able to focus on a specific kind of Galois connections relevant for property preservation; the lattice of predicates ordered by implication. This lattice is for reasons of simplicity usually represented as the

power set lattice $\mathbb{P}A$ ordered by the subset relation. The latter correspondence is due to the fact that in general predicates may be equivalently seen as sets of elements for which a predicate holds. This view is the basic idea used for predicate transformers.

$$\begin{pmatrix} p : A \to bool \\ p\ x \end{pmatrix} \equiv \begin{pmatrix} p : \mathbb{P}A \\ x \in p \end{pmatrix}$$

It enables the use of functions over sets, so-called predicate transformers, to relate abstract and concrete levels of specification. Actually, in Isabelle/HOL the definition of sets is given by identifying them with typed predicates.

This comes in neatly to solve the problem of relating the formalization of Galois connection with predicates. We define the type of predicate transformers as a special case of Galois connections over the lattice of sets using the following type abbreviation.

```
types (α, β) PT = (α set, β set) GC
```

Property preservation is now first expressed following Chou [Chou96] as a predicate which we call Ω

```
Ω R p q == ∀ (x,y) ∈ R. x ∈ p --> y ∈ q
```

The idea behind Ω is that `R` is a relation between concrete and abstract specification that preserves a pair of properties `(p, q)`.

To integrate the relational view with the predicate transformer view given by Galois connections of abstract and concrete predicate spaces, we first define the classical predicate transformers *sp* and *wp* representing strongest postcondition and weakest precondition.

```
sp R  == λ p. R `` p
wp R  == λ q. -(R⁻¹ ``(- q))
```

The operator `` denotes the image of a function applied to a set and - is the set complement. The pair `(sp R, wp R)` constitutes a Galois connection [MMS02]. This can now be shown as

```
(sp R, wp R) ∈ GC
```

where `GC` is the defining set of the type definition of Galois connections, named `GC` as well[2]. Once this property is shown it gives rise to the definition of a map from relations to Galois connections

```
GCofR == λ R. Abs_GC (sp R, wp R)
```

where `Abs_GC` is the internal HOL function that comes with the type definition of `GC` mapping each element of the defining set to its corresponding element of the new type `GC`. For the other direction we have to consider how to derive

[2] This is possible as Isabelle supports overloading.

the relation that is implicit in a predicate transformer represented by a Galois connection. We use the fact (c.f. [L+95, Proposition 7])

$$(a, b) \in R \equiv b \in \varepsilon\{a\}$$

in the following definition of the inverse map.

```
RofGC == (λ (e,p). {(a,b). b ∈ e {a}}) o Rep_GC
```

The function `Rep_GC` is the inverse to `Abs_GC`.

The correspondence between the Galois connection view and the relational view is described by the theorem

Theorem 1.

$$(sp\ R\ p \sqsubseteq q) \equiv (\Omega\ R\ p\ q) \equiv (p \sqsubseteq wp\ R\ q)\,.$$

We have proved this property in our formalization.

However, it is still not satisfactory as we would like to express the correspondence between relational view and predicate transformer view more explicitely using the notions of ε and π characteristic for Galois connection for the predicate transformer. Hence, as a final step to smooth over the integration of the two views, we overload the function symbols ε and π as selectors for the type `PT` of predicate transformers and give them the semantics we may according to the theorems proved above.

```
ε :: "[(α, β) PT, α set] => β set"
 "ε G == sp (RofGC G)"

π :: "[(α, β) PT, β set] => α set"
 "π G == wp (RofGC G)"
```

Finally, the proof of the correspondence given in Theorem 1 may be restated in Isabelle as

```
(ε G p ⊑ q) = (Ω (RofGC G) p q)
(Ω (RofGC G) p q) = (p ⊑ π G q)
```

So far we have only treated abstractly with property preservation. Concrete property preservation theorems can be expressed as instantiations of the presented general framework.

Property preservation is a general feature that in principle consists of one major theorem that has to be proved for a type of specific (reactive) system and a related temporal logic. For example, property preservation for CTL* state formulas Φ is expressed in terms of the relational view

$$\forall (s_1, s_2) \in R.\ s_1 \models \Phi \Rightarrow s_2 \models \Phi$$

where s_1, s_2 are states of the two related system specifications, and $\models$ is a satisfaction relation of CTL* (the model M is implicit.). Clearly, this is an instantiation of Ω. The general property preservation theorems one needs for the

application of abstraction in the setup of a certain model of reactive systems are then of the form (again for CTL*)

$$R \text{ is a simulation } \wedge R \text{ preserves } P_{\mathrm{CTL}^*} \Rightarrow R \text{ preserves CTL}^*$$

where P_{CTL^*} is the set of primitive formulas of CTL*. The notion of simulation based on relations is the standard one. It is awkward as it is on the level of elements related by R. The equivalent formulation of simulation in terms of Galois connections in comparison is much more concise:

$$(\varepsilon, \pi) \text{ is a simulation iff } \varepsilon \circ wp\ t_C \circ \pi \subseteq wp\ t_A$$

for all transition relations t_C and t_A of concrete and abstract system.

The equivalent formulations of the preservation properties in terms of Galois connections (c.f. [L$^+$95]), however, are not as simple as the ones expressed with Ω seen above.

That is, even if we just look at the defintion, each view has its advantages. When applying the framework to abstraction and refinement we intend to use each view where it is best suited. Concrete instantiations will be considered in the next section where the planned applications of the presented general framework are described.

4 Application to Abstraction and Refinement

In this section we briefly introduce two possible applications of the formalization of Galois connections and relations.

4.1 Abstraction for Statecharts

The formalization of statecharts by hierarchical automata [HK01] has been extended recently by data spaces. Thereby, infinite state spaces are possible as the data may range over infinite domains, like integers. Consequently, the need to construct abstractions that preserve important system properties arises.

To achieve this goal, further steps have been taken. A formulation of a branching time logic for the labelled transition systems representing the hierarchical automata is on the way. A suitable fragment of the μ-calculus has already been formalized from first principles and is going to be presented elsewhere. Once the formalization of the temporal logic is finished property preservation results for this logic have to be established. The derivation of the general property preservation theorems will be the first application of the framework for the statecharts formalization.

In our approach statecharts are translated into a model checkable form. To that end abstraction has to be employed. It is planned to devise construction algorithms similar to the ones presented in [SS99] to calculate abstracted forms of the charts that may then be passed on to a model checker. Based on the concepts for property preserving abstraction [CC77], Saïdi and Shankar describe

in [SS99] an algorithm as a proof routine for PVS that supports the automatic generation of abstractions for labelled transition systems. An implementation of similar algorithm for statecharts in Isabelle is planned. However, in order to formally verify that the calculated abstraction is indeed property preserving the present framework will be used. Although the construction algorithm produces abstractions it does not generally produce the right ones — but this can then be asserted within our framework. Using the general property preservation theorems, this task can be made feasible: when the algorithm produces a concrete instance this just needs to be checked. Due to general preservation theorems, the check is reduced to showing simulation based on Galois connection theory and preservation of primitive sub-formulas for the generated abstraction possibly employing the simpler representation with relations.

4.2 Construction of Refinements

Another possible application lies in the construction of refinements [HKS02]. Here, we intend to support a method that starts from predicate translations of specific system properties from an abstract system to an implementation. The goal is to derive a retrieve relation between abstract and concrete system based on the predicate translations. The motivation for this work is that in formal system development the system engineer usually has a rather predicative view. That is, he may well have a clear idea how certain properties, like invariants, preconditions and postconditions of certain operations translate from abstract specification to implementation, but it is difficult for him to construct the resulting data relation, i.e. the retrieve relation on abstract and concrete data space.

We have already explored the theory necessary for that method to some extent[HKS02]. In our approach we define a notion of ρ-closedness of predicate pairs that are input to the method. This criteria checks whether the predicate pairs respect the relation under construction. Given sufficient information in form of suitable pairs of predicates, preferably preconditions and postconditions of operations, it is possible to gain enough information to derive the retrieve relation. However, we do not know yet if it is possible to find general results describing whether the input information is sufficient.

Hence, it seems likely that an inductive procedure will result that enables the stepwise construction of the refinement relation as information is added. The predicate translations represent partial information about a predicate transformer. Using the correspondence between Galois connection and relations we intend to use the current framework to devise a set of Isabelle rules to inductively define the retrieve relation adding more and more information to its definition. The task of the framework will be primarily to avoid inconsistencies and introduce generalizations from pointwise information about the relation.

5 Conclusions

In this paper we have presented a general formulation of Galois connections and property preservation that is sufficiently abstract to be applied as a framework to other Isabelle formalizations. We have furthermore sketched a couple of such possible applications.

Theoretically, this work adds nothing new. All the correspondences and equivalences are well known, see for example Chou's work and the paper [L+95]. The conceptual idea to exploit the known equivalences between Galois connections and relations to get the best out of both worlds is original. Instead of postulating one view as the superior, we integrate the equivalence from start in order to switch between the views as suits us best.

The evaluation expected from the intended application domains will show whether much is gained by that dual approach. A clear indication for a synergetic effect is that most of the work done on abstract interpretation, which is the relevant approach for the statecharts application, has been based on Galois connections, whereas basically all refinement theory is based on a relational view. In our application domains we always have a bit of both worlds: the statecharts contain data spaces whose abstraction is clearly based on a data relation, whereas the abstraction of the reactive system itself is based on Galois connections. The construction of refinements starts — in difference to classical approaches, like for example refinement in Z — from a predicate translation view. Hence, at least for our interests we expect the combination presented here to be a suitable framework.

References

[BKS00] J. Burghardt, F. Kammüller and J. W. Sanders. *Isomorphisms of Galois Embeddings.* GMD Report 122, 2000.

[BKS01] J. Burghardt, F. Kammüller and J. W. Sanders. *On the Antisymmetry of Galois Embeddings.* Information Processing Letters, **79**:2, Elsevier, June 2001.

[Chou96] Ching-Tsun Chou. *Simple Proof Techniques for Property Preservation via Simulation.* Information Precessing Letters, **60**:129–134, Elsevier, 1996.

[Chou95] Ching-Tsun Chou. *Mechanical Verification of Distributed Algorithms in Higher-Order Logic.* The Computer Journal, 1995.

[CC77] P. Cousot and R. Cousot. *Abstract Interpretation.* Principles of Programming Languages, ACM Press, 1977.

[DP02] R. A. Davey and H. A. Priestley. *Introduction to Lattices and Order, Second Edition.* Cambridge University Press, 2002.

[Gl01] J. Glimming. *Logic and Automation for Algebra of Programming.* Masters Thesis, University of Oxford, 2001.

[HK01] S. Helke and F. Kammüller. *Representing Hierarchical Automata in Isabelle/HOL.* Theorem Proving in Higher Order Logics, Springer LNCS, 2001.

[HKS02] S. Helke, F. Kammüller, and J. W. Sanders. *Synthesizing Refinements from Predicate Translations.* Dependability of Component-based Systems. Dagstuhl-Seminar, 2002.

[L+95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. *Property Preserving Abstraction for the Verification of Concurrent Systems.* Formal Methods in System Design, **6**: 1–36, Kluwer 1995.

[MMS02] A. K. McIver, C. Morgan, and J. W. Sanders. *Programs and Abstraction.* Lecture Notes, University of Oxford, 2002.

[MSS86] A. Melton, D. A. Schmidt, and G. E. Strecker. *Galois connections and Computer Science applications*, Springer LNCS, **240**:299–312, 1986.

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, Springer LNCS, **828**, 1994.

[SS99] Hassen Saïdi and Natarajan Shankar. Abstract and Model Check While You Prove. In N. Halbwachs and D. Peled, editors, *11th International Conference on Computer Aided Verification, CAV'99, Springer LNCS*, **1633**, 1999.

Formalizing Abstract Algebra in Type Theory with Dependent Records

Xin Yu[1], Aleksey Nogin[1], Alexei Kopylov[2], and Jason Hickey[1]

[1] Department of Computer Science, California Institute of Technology
M/C 256-80, Pasadena, CA 91125, USA
[2] Department of Computer Science, Cornell University, Ithaca, NY 14853, USA

Abstract. Our goal is to develop a general formalization of abstract algebra suitable for a general reasoning. One of the most common ways to formalize abstract algebra is to make use of a module system to specify an algebra as a theory. However, this approach suffers from the fact that modules are usually not first-class objects in the formal system. In this paper, we develop a new approach based on the use of dependent record types. In our account, all algebraic structures are first-class objects, with the natural subtyping properties due to record extension (for example, a group is a subtype of a monoid). Our formalization cleanly separates the axiomatization of the algebra from its typing properties, corresponding more closely to a textbook presentation.

1 Introduction

The notions of abstract algebra are central to many areas of mathematics. Abstract algebra has also made many contributions to computer science, including abstract data types and object-oriented programming. Therefore we believe that having an *easily accessible* formalization of the abstract algebra notions in a formal system could be of great value. Formalization of many areas of mathematics could be based on such abstract algebra theory; and formalization of many computer science concepts could be modeled after it.

In this paper we explore a formalization of the abstract algebra concepts in type theory, based on the notion of an extensible record type. This is a part of larger effort to explore different approaches to formalizing basic abstract algebra concepts to find out which approach works the best. In particular, we want to find a formalization that is as close as possible to the standard text–book exposition of abstract algebra and corresponds closely to our intuition.

The group concept is the most central of all algebraic concepts; indeed, the techniques employed in group theory have long served as a pattern for other topics in algebra [4]. Therefore it is not surprising that the axiomatization of the abstract algebra concepts presented in this paper is centered around the axiomatization of the group and axiomatizations of other abstract algebra notions are very similar to the one of the group.

Of course, we are far from being the first ones to work with abstract algebra or group theory in a formal system. For example, Gunter working with HOL [7]

has proved group isomorphism theorems and shown the integers mod n to be an implementation of abstract groups [8]. Jackson has implemented computational abstract algebra in the NuPRL system [3,13,14]. And in IMPS [5] there is a notion of *little theories* [6] which they use for proving theorems about groups and rings.

One difference is that most, if not all, previous efforts concentrated on proving a specific "big theorem" and their main criterion for picking a certain axiomatization of basic concepts was to simplify the proof of that theorem. In contrast, in this paper we are primarily concerned with creating an axiomatization that would be natural and easily accessible to the widest possible range of theorems and applications.

Kammüller and Paulson [15] have proved Sylow's theorem in Isabelle-HOL, a large proof that required mechanizing a great deal of group theory. As Kammüller and Paulson state, one of the most common ways to formalize algebra is to make use of a module system, specifying the group as a theory. The main drawback of this approach is that modules are not first-class objects: it is difficult to quantify over them. To address this problem, Kammüller and Paulson proposed the development of a "section" concept, which would be like a module, but would be a first-class object that would capture only the formal parts of that module. In this paper, we propose the use of dependent record types to address this specific problem. In our formalization, records are used for providing a first-class formalization of algebraic objects, while also providing properties like inheritance and subtyping.

2 MetaPRL and Dependent Records

2.1 MetaPRL

MetaPRL [9, 12] is the latest system in the PRL family of theorem provers. The MetaPRL system combines the properties of an interactive LCF-style tactic-based proof assistant, a logical programming environment, and a formal methods programming toolkit. MetaPRL is also a logical framework that allows for reasoning in different logical theories. Its most extensively developed and most frequently used theory is a variation of the NuPRL intuitionistic type theory [3] (which in turn is based on the Martin-Löf type theory [17]).

2.2 Dependent Records

Records are tuples of labeled fields. We use the following syntax for records:

$$\{x_1 = a_1; \ldots; x_n = a_n\}$$

where $x_1, \ldots, x_n$ are labels of the string type and $a_1, \ldots a_n$ are corresponding fields. Each field of a record may have its own type. If each a_i has type A_i, then the above record has the type

$$\{x_1 \colon A_1; \ldots; x_n \colon A_n\}\,.$$

Reduction rules $\{r; x = a\}.x \longrightarrow a$ $\qquad$ $\{r; y = a\}.x \longrightarrow r.x$ when $x \neq y$
In particular: $\{x_1 = a_1; \ldots; x_n = a_n\}.x_i \longrightarrow a_i$ when all x_i's are distinct.

Type formation

$$\frac{\Gamma \vdash R_1 \text{ Type} \qquad \Gamma; \mathit{self}\colon R_1 \vdash R_2[\mathit{self}] \text{ Type}}{\Gamma \vdash \{R_1; R_2[\mathit{self}]\} \text{ Type}}$$

Introduction (membership rules)

$$\frac{\Gamma \vdash r \in R_1 \qquad \Gamma \vdash r \in R_2[r] \qquad \Gamma \vdash \{R_1; R_2[\mathit{self}]\} \text{ Type}}{\Gamma \vdash r \in \{R_1; R_2[\mathit{self}]\}}$$

Elimination (inverse typing rules)

$$\frac{\Gamma \vdash r \in \{R_1; R_2[\mathit{self}]\}}{\Gamma \vdash r \in R_1 \qquad \Gamma \vdash r \in R_2[r]}$$

Table 1. Inference rules for dependent records

In *dependent records* (or more formally dependently typed records) the type of fields may depend on values of the previous fields. In general a dependent record type has the following form

$$\{x_1\colon A_1; x_2\colon A_2\,[x_1]\,; \ldots; x_n\colon A_n\,[x_1, \ldots, x_{n-1}]\}\,.$$

Formally speaking, in MetaPRL we have two operators for constructing records:

- the constant $\{\}$ for the empty record;
- the ternary operation $\{r; x = a\}$ for the record extension.

We also have two constructors to form record types:

- the constant $\{\}$ for the empty record type (that contains *all* records);
- the ternary operation $\{\mathit{self}\colon R; x : A[\mathit{self}]\}$.

Here variable *self* is bounded in A. It refers to the record itself as a record of the type R. For example we may write something like

$$\{\mathit{self}\colon\ \{x\colon A\}\,; y : B\,[\mathit{self}.x]\}\,.$$

When we use the name "self" for this variable, we can use the shortening $\{R; A\,[\mathit{self}]\}$ for this type. Also we omit "*self.*" in the body of A, e.g. instead of $\{R; A\,[\mathit{self}.x; \mathit{self}.y]\}$ we write just $\{R; A\,[x, y]\}$. We assume that the ternary operations are left associative. For example, we write $\{x\colon A; y\colon B; z\colon C\}$ instead of $\{\{\{\{\}\,; x\colon A\}\,; y\colon B\}\,; z\colon C\}$.

There is also a binary operation $r.x$ for field selection, such that

$$\{x_1 = a_1; \ldots; x_n = a_n\}.x_i \leftrightarrow a_i.$$

In MetaPRL dependent records are defined using another primitive operation: dependent intersection [16]. The basic rules of our record calculus are shown in Table 1.

3 Formalization of Group Theory

The record calculus presented above gives us a natural way of formalizing the basic group theory definitions. It allows us to formalize the type restrictions on group operators (which are often kept implicit in mathematical textbooks) separately from the formalization of the actual axioms.

3.1 Formalization of Group-like Objects

Groupoid With dependent record, we formalize groupoid as:

$$\mathit{Groupoid}_i \leftrightarrow \{\mathrm{Car}\colon \mathbb{U}_i; *\colon \mathrm{Car} \to \mathrm{Car} \to \mathrm{Car}\}.$$

Semigroup Since a semigroup is a groupoid in which the binary operation is associative, we use a predicate to describe the constraint:

$$\mathit{isSemigroup}(g) \leftrightarrow \forall x\colon \mathrm{Car}_g.\forall y\colon \mathrm{Car}_g.\forall z\colon \mathrm{Car}_g.(x *_g y) *_g z = x *_g (y *_g z) \in \mathrm{Car}_g.$$

Then the set of semigroups can be defined as

$$\mathit{Semigroup}_i \leftrightarrow \{g\colon \mathit{Groupoid}_i \mid \mathit{isSemigroup}(g)\}.$$

Note that $\{x\colon A \mid P[x]\}$ is a standard type constructor for "set" in the NuPRL type theory, the elements of which are those elements $x \in A$ where the proposition $P[x]$ is true.

Obviously we can derive that a semigroup is also a groupoid.

Monoid A monoid is a semigroup with an identity element. So we first extend the dependent record $\mathit{Groupoid}_i$ with an extra field - the identity field,

$$\mathit{Premonoid}_i \leftrightarrow \{\mathit{Groupoid}_i; 1\colon \mathrm{Car}\},$$

and then define a predicate describing the constraint for a monoid

$$\mathit{isMonoid}(g) \leftrightarrow \mathit{isSemigroup}(g) \wedge \forall x\colon \mathrm{Car}_g.(1_g *_g x = x \in \mathrm{Car}_g \wedge x *_g 1_g = x \in \mathrm{Car}_g).$$

Then the set of monoids can be defined as

$$\mathit{Monoid}_i \leftrightarrow \{g\colon \mathit{Premonoid}_i \mid \mathit{isMonoid}(g)\}.$$

From our definition, a monoid is also a semigroup.

3.2 Formalization of Groups

For groups, we extend $Premonoid_i$ with an "inverse operation" field,

$$Pregroup_i \leftrightarrow \{Premonoid_i; \mathrm{inv}\colon \mathrm{Car} \to \mathrm{Car}\}\,,$$

and define the constraints for a group

$$isGroup(g) \leftrightarrow isSemigroup(g) \wedge \forall x\colon \mathrm{Car}_g.(1_g *_g x = x \in \mathrm{Car}_g) \wedge \\ \forall x\colon \mathrm{Car}_g.(x_g^{-1} *_g x = 1_g \in \mathrm{Car}_g)$$

where x_g^{-1} is the pretty-printed format of $\mathrm{inv}_g x$.

Then similar as the set of semigroups and the set of monoids, the set of groups is defined as

$$Group_i \leftrightarrow \{g\colon Pregroup_i \mid isGroup(g)\}\,.$$

The corresponding inference rules for groups are presented in Table 2.

Theorem 1. *All rules in Table 2 are derivable from the definitions given above.*

Proof. All these rules were derived in the MetaPRL system.

Now we can prove a set-and-binary-operation combination, for example, $\langle \mathbb{Z}, + \rangle$, is a group, i.e.,

$$\{\mathrm{Car} = \mathbb{Z}; * = \lambda x.\lambda y.(x + y); 1 = 0; \mathrm{inv} = \lambda x.(-x)\} \in Group_i$$

by applying the *group introduction* rule and then the *Pregroup introduction* and *isGroup introduction* rules.

Many properties of groups are immediate under the elimination rules, like those in Table 3.

We have also proved many useful theorems about groups in MetaPRL. For example, the left inverse/identity is also the right inverse/identity; a group is also a monoid; the identity is unique; the inverse operation is unique; for any $g \in Group_i$ and $a, b, c \in \mathrm{Car}_g$, $a *_g b = a *_g c \in \mathrm{Car}_g$ implies $b = c \in \mathrm{Car}_g$, and $(a *_g b)_g^{-1} = b_g^{-1} *_g a_g^{-1} \in \mathrm{Car}_g$.

In MetaPRL, these theorems are proved in a straightforward way. The basic idea is similar to that done by hand, but since MetaPRL provides some automated reasoning, some proofs might be easier. For illustration, we present a proof of one of the theorems.

Suppose we have already proved that the left inverse is also the right inverse, and now we want to prove the left identity is also the right identity

$$\frac{\Gamma \vdash g \in Group_i \qquad \Gamma \vdash a \in \mathrm{Car}_g}{\Gamma \vdash a *_g 1_g = a \in \mathrm{Car}_g}.$$

Our idea for proving it is

$$a *_g 1_g = a *_g (a_g^{-1} *_g a) = (a *_g a_g^{-1}) *_g a = 1_g *_g a = a,$$

Type formation

Pregroup
$$\frac{}{\Gamma \vdash \mathit{Pregroup}_i \ \mathrm{Type}}$$

isGroup
$$\frac{\Gamma \vdash \mathrm{Car}_g \ \mathrm{Type} \quad \Gamma; x\colon \mathrm{Car}_g; y\colon \mathrm{Car}_g \vdash x *_g y \in \mathrm{Car}_g \quad \Gamma \vdash 1_g \in \mathrm{Car}_g \quad \Gamma; x\colon \mathrm{Car}_g \vdash x_g^{-1} \in \mathrm{Car}_g}{\Gamma \vdash \mathit{isGroup}(g) \ \mathrm{Type}}$$

Group
$$\frac{}{\Gamma \vdash \mathit{Group}_i \ \mathrm{Type}}$$

Introduction

Pregroup
$$\frac{\Gamma \vdash g \in \{\mathrm{Car}\colon \mathbb{U}_i; *\colon \mathrm{Car} \to \mathrm{Car} \to \mathrm{Car}; 1\colon \mathrm{Car}; \mathrm{inv}\colon \mathrm{Car} \to \mathrm{Car}\}}{\Gamma \vdash g \in \mathit{Pregroup}_i}$$

isGroup
$$\frac{\Gamma \vdash \mathrm{Car}_g \ \mathrm{Type} \quad \Gamma \vdash \mathit{isSemigroup}(g) \quad \Gamma; x : \mathrm{Car}_g \vdash 1_g *_g x = x \in \mathrm{Car}_g \quad \Gamma; x\colon \mathrm{Car}_g \vdash x_g^{-1} *_g x = 1_g \in \mathrm{Car}_g}{\Gamma \vdash \mathit{isGroup}(g)}$$

Group
$$\frac{\Gamma \vdash g \in \mathit{Pregroup}_i \qquad \Gamma \vdash \mathit{isGroup}(g)}{\Gamma \vdash g \in \mathit{Group}_i}$$

Elimination

Pregroup
$$\frac{\Gamma; g : \{\mathrm{Car}\colon \mathbb{U}_i; *\colon \mathrm{Car} \to \mathrm{Car} \to \mathrm{Car}; \mathrm{inv}\colon \mathrm{Car} \to \mathrm{Car}\}; \Delta[g] \vdash C[g]}{\Gamma; g : \mathit{Pregroup}_i; \Delta[g] \vdash C[g]}$$

isGroup
$$\frac{\begin{array}{c}\Gamma; u\colon \mathit{isGroup}(g); w\colon \forall x\colon \mathrm{Car}_g.(1_g *_g x = x \in \mathrm{Car}_g); \\ v\colon \forall x\colon \mathrm{Car}_g.\forall y\colon \mathrm{Car}_g.\forall z\colon \mathrm{Car}_g.(x *_g y)*_g = x *_g (y *_g z) \in \mathrm{Car}_g; \\ y\colon \forall x\colon \mathrm{Car}_g.(x_g^{-1} *_g x = 1_g \in \mathrm{Car}_g); \Delta[u] \vdash C[u]\end{array}}{\Gamma; u\colon \mathit{isGroup}(g); \Delta[u] \vdash C[u]}$$

Group
$$\frac{\Gamma; g\colon \mathit{Pregroup}_i; u\colon \mathit{isGroup}(g); \Delta[g] \vdash C[g]}{\Gamma; g\colon \mathit{Group}_i; \Delta[g] \vdash C[g]}$$

Table 2. Inference rules for groups

Membership

$$\frac{\Gamma \vdash g \in \mathit{Group}_i}{\Gamma \vdash \mathrm{Car}_g \in \mathbb{U}_i} \qquad \frac{\Gamma \vdash g \in \mathit{Group}_i}{\Gamma \vdash *_g \in \mathrm{Car}_g \to \mathrm{Car}_g \to \mathrm{Car}_g}$$

$$\frac{\Gamma \vdash g \in \mathit{Group}_i}{\Gamma \vdash 1_g \in \mathrm{Car}_g} \qquad \frac{\Gamma \vdash g \in \mathit{Group}_i}{\Gamma \vdash \mathrm{inv}_g \in \mathrm{Car}_g \to \mathrm{Car}_g}$$

Associativity

$$\frac{\Gamma \vdash g \in \mathit{Group}_i \quad \Gamma \vdash a \in \mathrm{Car}_g \quad \Gamma \vdash b \in \mathrm{Car}_g \quad \Gamma \vdash c \in \mathrm{Car}_g}{\Gamma \vdash (a *_g b) *_g c = a *_g (b *_g c) \in \mathrm{Car}_g}$$

Left identity and left inverse

$$\frac{\Gamma \vdash g \in \mathit{Group}_i \quad \Gamma \vdash a \in \mathrm{Car}_g}{\Gamma \vdash 1_g *_g a = a \in \mathrm{Car}_g} \qquad \frac{\Gamma \vdash g \in \mathit{Group}_i \quad \Gamma \vdash a \in \mathrm{Car}_g}{\Gamma \vdash x_g^{-1} *_g a = 1_g \in \mathrm{Car}_g}$$

Table 3. Some simple group properties

where the second equation holds because of the associativity of $*_g$ and the third holds because the left inverse is also the right inverse.

To prove it in the MetaPRL proof editor, we first replace 1_g with $a_g^{-1} *_g a$, which can be done by a tactic `substT` provided by MetaPRL. The usage is `substT` $(a = b \in A)$ `i`, which replaces all occurrences of the term a with b in clause i ($i = 0$ implies the conclusion). So we navigate to this rule and apply the `substT` $(1_g = a_g^{-1} *_g a \in \mathrm{Car}_g)$ `0 thenT autoT` tactic.[3]

Two subgoals are generated. The first one,

$$\frac{\Gamma \vdash g \in \mathit{Group}_i \qquad \Gamma \vdash a \in \mathrm{Car}_g}{\Gamma \vdash 1_g = a_g^{-1} *_g a \in \mathrm{Car}_g},$$

is trivial since we have

$$\frac{\Gamma \vdash g \in \mathit{Group}_i \qquad \Gamma \vdash a \in \mathrm{Car}_g}{\Gamma \vdash a_g^{-1} *_g a = 1_g \in \mathrm{Car}_g}$$

and $=$ is symmetric. So, by applying the `equalSymT thenT autoT` tactic, this subgoal is proved.

As for the second subgoal,

$$\frac{\Gamma \vdash g \in \mathit{Group}_i \qquad \Gamma \vdash a \in \mathrm{Car}_g}{\Gamma \vdash a *_g (a_g^{-1} *_g a) = a \in \mathrm{Car}_g},$$

we can utilize the associativity property of groups (listed in Table 3) by applying the tactic `substT` $(a *_g (a_g^{-1} *_g a) = (a_g^{-1} *_g a) *_g a \in \mathrm{Car}_g)$ `0 thenT autoT`, which

[3] The `autoT` tactic performs "automated" proving based on repeated application of several "basic" tactics; and the infix function `thenT` is a tactical used for sequencing [10]: the proof first applies the substitution, and then applies the `autoT` tactic.

generates two new subgoals

$$\frac{\Gamma \vdash g \in \mathit{Group}_i \qquad \Gamma \vdash a \in \mathrm{Car}_g}{\Gamma \vdash a *_g (a_g^{-1} *_g a) = (a_g^{-1} *_g a) *_g a \in \mathrm{Car}_g}$$

and

$$\frac{\Gamma \vdash g \in \mathit{Group}_i \qquad \Gamma \vdash a \in \mathrm{Car}_g}{\Gamma \vdash (a *_g a_g^{-1}) *_g a = a \in \mathrm{Car}_g}.$$

The first one can be eliminated with the use of `equalSymT`. In the second one $a *_g a_g^{-1}$ can be replaced with 1_g thanks to the right inverse property we have proved. After this substitution, we get the subgoal of proving $1 *_g a = a$, trivial by the left identity property (listed in Table 3). This completes the proof of this theorem.

For all the theorems proved as well as their proofs, see [11].

3.3 Formalization of Abelian Groups

If we define a predicate

$$\mathit{isCommut}(g) \leftrightarrow \forall x\colon \mathrm{Car}_g.\forall y\colon \mathrm{Car}_g.(x *_g y = y *_g x \in \mathrm{Car}_g),$$

then commutative semigroups, commutative monoids, and abelian groups can be defined respectively as

$$\mathit{Csemigroup}_i \leftrightarrow \{g\colon \mathit{Semigroup}_i \mid \mathit{isCommut}(g)\};$$

$$\mathit{Cmonoid}_i \leftrightarrow \{g\colon \mathit{Monoid}_i \mid \mathit{isCommut}(g)\};$$

$$\mathit{Abelg}_i \leftrightarrow \{g\colon \mathit{Group}_i \mid \mathit{isCommut}(g)\}.$$

Their type formation, introduction, and elimination rules are simple. For example, for Abelg_i, we have

$$\frac{}{\Gamma \vdash \mathit{Abelg}_i \text{ Type}} \qquad \frac{\Gamma \vdash g \in \mathit{Group}_i \qquad \Gamma \vdash \mathit{isCommut}(g)}{\Gamma \vdash g \in \mathit{Abelg}_i}$$

$$\frac{\Gamma; g\colon \mathit{Group}_i; u\colon \mathit{isCommut}(g); \Delta[g] \vdash C[g]}{\Gamma; g\colon \mathit{Abelg}_i; \Delta[g] \vdash C[g]}$$

where

$$\frac{\Gamma \vdash \mathrm{Car}_g \text{ Type} \qquad \Gamma; x, y\colon \mathrm{Car}_g \vdash x *_g y \in \mathrm{Car}_g}{\Gamma \vdash \mathit{isCommut}(g) \text{ Type}} \qquad \frac{\Gamma \vdash \mathrm{Car}_g \text{ Type} \qquad \Gamma; x, y\colon \mathrm{Car}_g \vdash x *_g y = y *_g x \in \mathrm{Car}_g}{\Gamma \vdash \mathit{isCommut}(g)}$$

$$\frac{\Gamma; u\colon \mathit{isCommut}(g); v\colon \forall x\colon \mathrm{Car}_g.\forall y\colon \mathrm{Car}_g.(x *_g y = y *_g x \in \mathrm{Car}_g); \Delta[u] \vdash C[u]}{\Gamma; u\colon \mathit{isCommut}(g); \Delta[u] \vdash C[u]}.$$

Apparently, if $g \in \mathit{Abelg}_i$ then $g \in \mathit{Group}_i$.

3.4 Formalization of Subgroups

First we define the generalized "subStructure" for all groupoids:

$$h \subseteq_{Str} g \leftrightarrow \mathrm{Car}_h \subseteq \mathrm{Car}_g \wedge *_g = *_h \in \mathrm{Car}_h \to \mathrm{Car}_h \to \mathrm{Car}_h.$$

Then, for example, the submonoid can be defined as

$$h \subseteq_{Monoid_i} g \leftrightarrow h \in Monoid_i \wedge g \in Monoid_i \wedge h \subseteq_{Str} g$$

and the subgroup

$$h \subseteq_{Group_i} g \leftrightarrow h \in Group_i \wedge g \in Group_i \wedge h \subseteq_{Str} g.$$

We proved that if $s \subseteq_{Group_i} g$, then

1. Car_s is closed under $*_g$;
2. $1_g = 1_s \in \mathrm{Car}_s$;
3. for all $a \in \mathrm{Car}_s$, $a_g^{-1} = a_s^{-1} \in \mathrm{Car}_s$.

We also proved the intersection of two subgroups is again a subgroup:

$$\frac{\Gamma \vdash s_1 \subseteq_{Group_i} g \qquad \Gamma \vdash s_2 \subseteq_{Group_i} g}{\Gamma \vdash \{\mathrm{Car} = \mathrm{Car}_{s_1} \cap \mathrm{Car}_{s_2}; * = *_g; 1 = 1_g; \mathrm{inv} = \mathrm{inv}_g\} \subseteq_{Group_i} g}.$$

3.5 Formalization of Cyclic Groups

The group power operation Suppose g is a group. For any element $a \in \mathrm{Car}_g$, we define

$$a_g^n = \begin{cases} \underbrace{a *_g a *_g \ldots *_g a}_{n} & \text{if } n > 0 \\ e_g & \text{if } n = 0 \\ \underbrace{a_g^{-1} *_g a_g^{-1} *_g \ldots *_g a_g^{-1}}_{-n} & \text{if } n < 0 \end{cases}$$

as the power operation of the group g based on a (a is the **base**).

We formalize it with mathematical induction as

$$a_g^n = \begin{cases} a *_g a_g^{n-1} & \text{if } n > 0 \\ e_g & \text{if } n = 0 \\ a_g^{-1} *_g a_g^{n+1} & \text{if } n < 0 \end{cases},$$

where n is of the integer type. By induction, if a is in Car_g, then a_g^n is in Car_g, and for any $m, n \in \mathbb{Z}$, $a_g^m *_g a_g^n = a_g^{m+n} \in \mathrm{Car}_g$ and $(a_g^m)_g^n = a_g^{m \times n} \in \mathrm{Car}_g$. Also, if $s \subseteq_{Group_i} g$, then for all $a \in \mathrm{Car}_s$ and $n \in \mathbb{Z}$, $a_g^n = a_s^n \in \mathrm{Car}_s$.

The cyclic group We define a predicate

$$isCyclic(g) \leftrightarrow \exists a\colon \mathrm{Car}_g.\forall x\colon \mathrm{Car}_g.\exists n\colon \mathbb{Z}.(x = a_g^n \in \mathrm{Car}_g)$$

to describe a group g is cyclic. Its inference rules are listed below:

$$\textit{Type formation}\ \frac{\Gamma \vdash g \in Group_i}{\Gamma \vdash isCyclic(g)\ \mathrm{Type}} \qquad \textit{Introduction}\ \frac{\Gamma \vdash g \in Group_i \quad \Gamma \vdash a \in \mathrm{Car}_g \quad \Gamma; x\colon \mathrm{Car}_g \vdash \exists n\colon \mathbb{Z}.x = a_g^n \in \mathrm{Car}_g}{\Gamma \vdash isCyclic(g)}$$

$$\textit{Elimination}\ \frac{\Gamma; x\colon isCyclic(g); a\colon \mathrm{Car}_g; \forall x\colon \mathrm{Car}_g.\exists n\colon \mathbb{Z}.(x = a_g^n \in \mathrm{Car}_g); \Delta[x] \vdash C[x]}{\Gamma; x\colon isCyclic(g); \Delta[x] \vdash C[x]}.$$

Every cyclic group is abelian. And every non-trivial subgroup of a cyclic group is cyclic.

The cyclic subgroup The cyclic subgroup of g generated by a is defined as

$$cycSubg(g; a) \leftrightarrow \left\{\mathrm{Car} = \left\{x\colon \mathrm{Car}_g \mid \exists n\colon \mathbb{Z}.x = a_g^n \in \mathrm{Car}_g\right\}; * = *_g; 1 = 1_g; \mathrm{inv} = \mathrm{inv}_g\right\}.$$

We have $g \in Group_i$ implies $cycSubg(g; a) \in Group_i$ and $cycSubg(g; a) \subseteq_{Group_i} g$.

3.6 Formalization of Cosets and Normal Subgroups

We define the left coset and the right coset as

$$Lcoset(s; g; b) \leftrightarrow \{x\colon \mathrm{Car}_g \mid \exists a : \mathrm{Car}_s.(x = b *_g a \in \mathrm{Car}_g)\},$$

$$Rcoset(s; g; b) \leftrightarrow \{x\colon \mathrm{Car}_g \mid \exists a : \mathrm{Car}_s.(x = a *_g b \in \mathrm{Car}_g)\},$$

and normal subgroup as

$$s \lhd_i g \leftrightarrow s \subseteq_{Group_i} g \wedge \forall x\colon \mathrm{Car}_g.Lcoset(s; g; x) =_e Rcoset(s; g; x)$$

where $=_e$ means extensional equality.

Then both cosets are subsets of the carrier of g

$$\frac{\Gamma \vdash s \subseteq_{Group_i} g \qquad \Gamma \vdash b \in \mathrm{Car}_g}{\Gamma \vdash Lcoset(s; g; b) \subseteq \mathrm{Car}_g} \qquad \frac{\Gamma \vdash s \subseteq_{Group_i} g \qquad \Gamma \vdash b \in \mathrm{Car}_g}{\Gamma \vdash Rcoset(s; g; b) \subseteq \mathrm{Car}_g}$$

and all subgroups of abelian groups are normal

$$\frac{\Gamma \vdash s \subseteq_{Group_i} g \qquad \Gamma \vdash isCommut(g)}{\Gamma \vdash s \lhd_i g}.$$

Type formation

isGroupHom
$$\frac{\begin{array}{c}\Gamma \vdash f \in \mathrm{Car}_h \to \mathrm{Car}_g \\ \Gamma \vdash \mathrm{Car}_h\ \mathrm{Type} \qquad \Gamma; x\colon \mathrm{Car}_h; y\colon \mathrm{Car}_h \vdash x *_h y \in \mathrm{Car}_h \\ \Gamma \vdash \mathrm{Car}_g\ \mathrm{Type} \qquad \Gamma; x\colon \mathrm{Car}_g; y\colon \mathrm{Car}_g \vdash x *_g y \in \mathrm{Car}_g\end{array}}{\Gamma \vdash \mathit{isGroupHom}(f; h; g)\ \mathrm{Type}}$$

GroupHom
$$\frac{\begin{array}{cc}\Gamma \vdash \mathrm{Car}_h\ \mathrm{Type} & \Gamma; x\colon \mathrm{Car}_h; y\colon \mathrm{Car}_h \vdash x *_h y \in \mathrm{Car}_h \\ \Gamma \vdash \mathrm{Car}_g\ \mathrm{Type} & \Gamma; x\colon \mathrm{Car}_g; y\colon \mathrm{Car}_g \vdash x *_g y \in \mathrm{Car}_g\end{array}}{\Gamma \vdash \mathit{GroupHom}(h; g)\ \mathrm{Type}}$$

Introduction

isGroupHom
$$\frac{\begin{array}{c}\Gamma \vdash \mathrm{Car}_h\ \mathrm{Type} \\ \Gamma; x\colon \mathrm{Car}_h; y\colon \mathrm{Car}_h \vdash f\ (x *_h y) = f\ x *_g f\ y \in \mathrm{Car}_g\end{array}}{\Gamma \vdash \mathit{isGroupHom}(f; h; g)}$$

GroupHom
$$\frac{\begin{array}{cc}\Gamma \vdash h \in \mathit{Group}_i & \Gamma \vdash g \in \mathit{Group}_i \\ \Gamma \vdash f \in \mathrm{Car}_h \to \mathrm{Car}_g & \Gamma \vdash \mathit{isGroupHom}(f; h; g)\end{array}}{\Gamma \vdash f \in \mathit{GroupHom}(h; g)}$$

Elimination

isGroupHom
$$\frac{\begin{array}{c}\Gamma; u\colon \mathit{isGroupHom}(f; h; g); \Delta[u]; \\ v\colon \forall x\colon \mathrm{Car}_h.\forall y\colon \mathrm{Car}_h. f\ (x *_h y) = f\ x *_g f\ y \in \mathrm{Car}_g \vdash C[u]\end{array}}{\Gamma; u\colon \mathit{isGroupHom}(f; h; g); \Delta[u] \vdash C[u]}$$

GroupHom
$$\frac{\Gamma; f\colon \mathrm{Car}_h \to \mathrm{Car}_g; u\colon \mathit{isGroupHom}(f; h; g); \Delta[f] \vdash C[f]}{\Gamma; f\colon \mathit{GroupHom}(h; g); \Delta[f] \vdash C[f]}$$

Table 4. Inference rules for group homomorphisms.

3.7 Formalization of Group Mappings

The group homomorphism We first define group homomorphisms:

$$\mathit{isGroupHom}(f;h;g) \leftrightarrow \forall x\colon \mathrm{Car}_h.\forall y\colon \mathrm{Car}_h.(f\ (x *_h y) = f\ x *_g f\ y \in \mathrm{Car}_g);$$

$$\mathit{GroupHom}(h;g) \leftrightarrow \{f\colon \mathrm{Car}_h \to \mathrm{Car}_g \mid \mathit{isGroupHom}(f;h;g)\}.$$

Their inference rules are in Table 4.

Group homomorphisms preserve group structure. Put differently, if f is a group homomorphism from h into g, we might know the structure of g from that of h. For example, f maps 1_h to 1_g, and maps a_h^{-1} to $(f\ a)_g^{-1}$ for $a \in \mathrm{Car}_h$; if s is a subgroup of h, then the image of s under f is a subgroup of g

$$\frac{\Gamma \vdash g \in \mathit{Group}_i \qquad \Gamma \vdash f \in \mathit{GroupHom}(h;g) \qquad \Gamma \vdash s \subseteq_{\mathit{Group}_i} h}{\Gamma \vdash \{\mathrm{Car} = \{x\colon \mathrm{Car}_g \mid \exists y\colon \mathrm{Car}_h.(x = f\ y \in \mathrm{Car}_g)\}; * = *_g; 1 = 1_g; \mathrm{inv} = \mathrm{inv}_g\} \subseteq_{\mathit{Group}_i} g};$$

if t is a subgroup of g, then the inverse image of t under f is a subgroup of h

$$\frac{\Gamma \vdash h \in \mathit{Group}_i \qquad \Gamma \vdash f \in \mathit{GroupHom}(h;g) \qquad \Gamma \vdash t \subseteq_{\mathit{Group}_i} g}{\Gamma \vdash \{\mathrm{Car} = \{x\colon \mathrm{Car}_h \mid f\ x \in \mathrm{Car}_t\}; * = *_h; 1 = 1_h; \mathrm{inv} = \mathrm{inv}_h\} \subseteq_{\mathit{Group}_i} h}.$$

We have proved all these properties of homomorphisms in MetaPRL.

Other group mappings Similarly, we can define group monomorphism, group epimorphism, and group isomorphism etc. as

$$\mathit{GroupMono}(h;g) \leftrightarrow \{f\colon \mathit{GroupHom}(h;g) \mid \mathit{Injective}(f;\mathrm{Car}_h;\mathrm{Car}_g)\},$$

$$\mathit{GroupEpi}(h;g) \leftrightarrow \{f\colon \mathit{GroupHom}(h;g) \mid \mathit{Surjective}(f;\mathrm{Car}_h;\mathrm{Car}_g)\},$$

$$h \cong g \leftrightarrow \{f\colon \mathit{GroupHom}(h;g) \mid \mathit{Bijective}(f;\mathrm{Car}_h;\mathrm{Car}_g)\},$$

where $\mathit{Injective}(f;\mathrm{Car}_h;\mathrm{Car}_g)$, $\mathit{Surjective}(f;\mathrm{Car}_h;\mathrm{Car}_g)$, $\mathit{Bijective}(f;\mathrm{Car}_h;\mathrm{Car}_g)$ means f is injective, surjective, and bijective from Car_h to Car_g respectively.

If f is a group epimorphism from h to g, then h is abelian implies g is abelian. If f is a group isomorphism, then its reverse mapping is also a group isomorphism.

3.8 Formalization of Group Kernels

We define the group kernel of a homomorphism f from h to g as

$$\mathit{GroupKer}(f;h;g) \leftrightarrow \{\mathrm{Car} = \{x\colon \mathrm{Car}_h \mid f\ x = 1_g \in \mathrm{Car}_g\}; * = *_h; 1 = 1_h; \mathrm{inv} = \mathrm{inv}_h\}.$$

Its introduction rule is

$$\frac{\Gamma \vdash h \in \mathit{Group}_i \qquad \Gamma \vdash g \in \mathit{Group}_i \qquad \Gamma \vdash f \in \mathit{GroupHom}(h;g)}{\Gamma \vdash \mathit{GroupKer}(f;h;g) \in \mathit{Group}_i}.$$

We proved if $h \in \mathit{Group}_i$, $g \in \mathit{Group}_i$, and $f \in \mathit{GroupHom}(f;h;g)$, then $\mathit{GroupKer}(f;h;g) \subseteq_{\mathit{Group}_i} h$; what's more, $\mathit{GroupKer}(f;h;g) \lhd_i h$ (because both cosets of $\mathit{GroupKer}(f;h;g)$ containing any $x \in \mathrm{Car}_h$ are equal to the set whose element has the same image under f as x). We also proved that f is a group monomorphism from h to g if and only if the kernel of f contains 1_h alone.

3.9 Formalization of Quotient Group

Traditionally, if G is a group and H is a normal subgroup of G, then the collection of all cosets of H in G together with the multiplication of cosets in the form

$$(aH)(bH) = abH \qquad\qquad a, b \in G$$

forms a group called *quotient group*, or *factor group*.

Equivalently the quotient group of G by H can also be defined as the set of collections of elements in G together with G's binary operation $*$, where two elements $a, b \in G$ are in the same collection if and only if $a * b^{-1} \in H$.

So we define the quotient group of g by h as

$$g/h \leftrightarrow \left\{\mathrm{Car} = \mathit{quot}\ x, y : \mathrm{Car}_g // x *_g y_g^{-1} \in \mathrm{Car}_h; * = *_g; 1 = 1_g; \mathrm{inv} = \mathrm{inv}_g\right\},$$

where the elements of $\mathit{quot}\ x, y : \mathrm{Car}_g // x *_g y_g^{-1} \in \mathrm{Car}_h$ are the elements of Car_g, but the equality is determined by the equivalence relation $x *_g y_g^{-1} \in \mathrm{Car}_h$.

Compared with specifying the quotient group as a set of cosets, our formalization using the quotient type has many advantages. If we use the former method, then first g/h will be in Group_{i+1}, not Group_i; second, it is difficult to define operations on cosets, but for quotient type, we have them for free (we only need to prove that they are well-formed); third, we will not be able to prove the *Fundamental Homomorphism Theorem* because for a given coset we will not be able to construct an element of this coset since there is no axiom of choice in intuitionistic type theory.

With our formalization, assuming $h \lhd_i g$, we have proved $g/h \in \mathit{Group}_i$, and if g is abelian, then g/h is abelian. Also, $\lambda x.x$ is an epimorphism of g to g/h with kernel h. We also proved the *First Isomorphism Theorem for Groups*, which states that if k is the kernel of a group epimorphism from g to h, h is isomorphism to the quotient group g/k.

4 Discussion, Conclusions, Future Work

We have shown an initial part of the formalization of abstract algebra that provides a general-purpose account using records to specify first-class algebraic objects. This formalization has shown itself to be quite natural compared with many other methods, and reasoning often corresponds closely to textbook accounts. We believe this methodology can be extended easily to more advanced algebraic objects.

4.1 Ring/Field Theory

We have not yet implemented any parts of ring and field theory; however we do believe that we already have a good understanding on how to proceed. The objective is to use a `rename` operation in the record theory that would allow renaming fields of record objects (but not record types). Then we could define

the ring type by simply extending the existing group theory types with extra fields. We could define, for example,

$$\mathit{Prering}_i \leftrightarrow \{\mathit{Groupoid}_i; + \colon \mathrm{Car} \to \mathrm{Car} \to \mathrm{Car}; 0 \colon \mathrm{Car}; \mathrm{neg} \colon \mathrm{Car} \to \mathrm{Car}\}$$

$$\mathrm{additive\_group}(r) \leftrightarrow \mathtt{rename}(+ \rightsquigarrow *, 0 \rightsquigarrow 1, \mathrm{neg} \rightsquigarrow \mathrm{inv})\{r\}$$

The proof automation for `rename` should be capable of easily establishing $\forall r : \mathit{Prering}_i.\mathrm{additive\_group}(r) \in \mathit{Pregroup}_i$. In turn, this would allow us to define

$$\begin{aligned}\mathit{isRing}(g) \leftrightarrow \quad & \mathit{isSemigroup}(g) \\ & \wedge \mathit{isGroup}(\mathrm{additive\_group}(r)) \wedge \mathit{isCommut}(\mathrm{additive\_group}(r)) \\ & \wedge \forall a, b, c \colon \mathrm{Car}_r.((a +_r b) *_r c = a *_r c +_r b *_r c \in \mathrm{Car}_r \wedge \\ & \qquad a *_r (b +_r c) = a *_r b +_r a *_r c \in \mathrm{Car}_r)\end{aligned}$$

and allow reusing all the existing group theorems for the additive group of a ring.

4.2 Comparison with the Formalization in CZF

We have also formalized group theory in MetaPRL CZF, which is based on Aczel's Constructive Zermelo-Fraenkel set theory [1,2]. In the CZF formalization, groups and other objects are specified as a set of rules [18, 19], where the parts of an object are labeled constants. This is adequate for formalizing many objects in abstract algebra; we can derive theorems about them, and we can also formalize specific objects. However, with this method, the formalized objects are not first-class, which makes it impossible to quantify over them. We would need first-class modules to make it work effectively, a feature not yet supported in MetaPRL. In addition, the formalization is awkward because typing axioms are not cleanly separated from the principal algebra axioms. In our type theory formalization, all objects are first-class and the type information is cleanly separated.

On the other hand, set theory is more natural in some cases. For example, types use intensional equality, but we often care more about extensional properties of algebraic objects. We are currently investigating extensional type-theoretic interpretations.

References

1. Peter Aczel. The type theoretic interpretation of constructive set theory: Inductive definition. In *Logic, Methodology and Philosophy of Science VII*, pages 17–49. Elsevier Science Publishers, 1986.
2. Peter Aczel and Michael Rathjen. Notes on constructive set theory. Technical Report 40, Mittag-Leffler, 2000/2001.
3. Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the NuPRL Development System*. Prentice-Hall, NJ, 1986.

4. Richard A. Dean. *Elements of Abstract Algebra*. Wiley, New York, 2nd edition, 1966.
5. William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: An interactive mathematical proof system. *Journal of Automated Reasoning*, 11:213–248, 1993.
6. William M. Farmer and F. Javier Thayer Joshua D. Guttman. Little theories. In D. Kapur, editor, *Automated-Deduction-CADE-11*, Lecture Notes in Artificial Intelligence, pages 567–581, New York, June 1992. Springer-Verlag.
7. Michael Gordon and Tom Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Oder Logic*. Cambridge University Press, Cambridge, 1993.
8. Elsa Gunter. Doing algebra in simple type theory. Technical Report MS-CIS-89-38, Logic & Computation 09, Department of Computer and Information Science, Moore School of Engineering, University of Pennsylvania, Jun 1989. Distributed with the HOL system in the directory Training/studies/intmod/doingalgpaper.
9. Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir N. Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu. MetaPRL — A modular logical environment. Accepted to the TPHOLs 2003 Conference, 2003.
10. Jason J. Hickey. *The MetaPRL Logical Programming Environment*. PhD thesis, Cornell University, Ithaca, NY, January 2001.
11. Jason J. Hickey, Brian Aydemir, Yegor Bryukhov, Alexei Kopylov, Aleksey Nogin, and Xin Yu. A listing of MetaPRL theories. `http://metaprl.org/theories.pdf`.
12. Jason J. Hickey, Aleksey Nogin, Alexei Kopylov, et al. MetaPRL home page. `http://metaprl.org/`.
13. Paul B. Jackson. Exploring abstract algebra in constructive type theory. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, volume 814 of *Lecture Notes in Artificial Intelligence*, pages 590–604, New York, June 1994. Springer-Verlag.
14. Paul B. Jackson. *Enhancing the NuPRL Proof Development System and Applying it to Computational Abstract Algebra*. PhD thesis, Cornell University, Ithaca, NY, January 1995.
15. F. Kammüller and L. C. Paulson. A formal proof of Sylow's first theorem – an experiment in abstract algebra with Isabelle HOL. *Journal of Automated Reasoning*, 23(3-4):235–264, 1999.
16. Alexei Kopylov. Dependent intersection: A new way of defining records in type theory. In *Proceedings of 18th IEEE Symposium on Logic in Computer Science*, 2003. To appear.
17. Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.
18. Xin Yu. Formalizing abstract algebra in constructive set theory. Master's thesis, California Institute of Technology, 2002.
19. Xin Yu and Jason J. Hickey. Formalizing abstract algebra in constructive set theory. Technical Report caltechCSTR2003.004, California Institute of Technology, Caltech, CA 91125, June 2003.

Implementing and Automating Basic Number Theory in MetaPRL Proof Assistant*

Yegor Bryukhov[1], Alexei Kopylov[2], Vladimir Krupski[3], and Aleksey Nogin[4]

[1] Graduate Center, City University of New York
365 Fifth Avenue, New York, NY 10016
ybryukhov@gc.cuny.edu
[2] Department of Computer Science, Cornell University, Ithaca, NY 14853
kopylov@cs.cornell.edu
[3] Laboratory for Logical Problems of Computer Science
Department of Mathematical Logic and Theory of Algorithms
Faculty of Mechanics and Mathematics, Moscow State University
Vorob'evy Gory, 119899 RUSSIA
krupski@lpcs.math.msu.ru
[4] Department of Computer Science, California Institute of Technology
M/C 256-80, Pasadena, CA 91125
nogin@cs.caltech.edu

Abstract. No proof assistant can be considered complete unless it provides facilities for basic arithmetical reasoning. Indeed, integer theory is a part of the necessary foundation for most of mathematics, logic and computer science. In this paper we present our approach to implementing arithmetic in the intuitionistic type theory of the MetaPRL proof assistant. We focus on creating an axiomatization that would take advantage of the computational features of MetaPRL type theory. Also, we implement the Arith decision procedure as a *tactic* that constructs proofs based on existing axiomatization, instead of being a part of the "trusted" code base.

1 Introduction

MetaPRL [4,6] is the latest system in the PRL family of theorem provers [2,3]. The MetaPRL system combines the properties of an interactive LCF-style tactic-based proof assistant, a logical programming environment, and a formal methods programming toolkit. MetaPRL is also a logical framework that allows for reasoning in different logical theories. Its most extensively developed and most frequently used theory is a variation of the NuPRL intuitionistic type theory [3] (which in turn is based on the Martin-Löf type theory [9]).

* This work was supported in part by the DoD Multidisciplinary University Research Initiative (MURI) program administered by the Office of Naval Research (ONR) under Grant N00014-01-1-0765, the Defense Advanced Research Projects Agency (DARPA), the United States Air Force, the Lee Center, and by NSF Grant CCR 0204193.

Since MetaPRL type theory is so close to NuPRL's one, it is natural to use NuPRL's implementation of arithmetic as a basis for comparison. In NuPRL, a big part of the support for arithmetical reasoning is provided via two decision procedures — Arith [1] and Sup-Inf [14]. As output, these decision procedures do not provide real proofs; instead they only tell if current goal is provable or not according to their knowledge. This approach is at least imperfect because it extends the code base we have to *trust*.

Including such trusted decision procedures in a system that allows formalizing different logical theories, as well as different variations of the same logical theory, can have additional disadvantages. It significantly reduces the flexibility — whenever we want to change or update some aspects of a logical theory being used in such a system, we have to make sure that all the assumptions made by all the trusted decision procedures used remain valid in the updated theory. In MetaPRL, we wanted to avoid using trusted decision procedures, turning them into tactics instead. This way even if such procedure is flawed, or is not fully compatible with the logical theory used, the worst thing could happen is that it will fail to prove the statement it was applied to (of course, we still have to trust our proof checker).

Another goal we had when designing the arithmetical theory for MetaPRL is efficiency. MetaPRL is a highly efficient system; on most proof tasks it is over two orders of magnitude faster than its predecessor, NuPRL. We wanted the new arithmetics implementation to keep up with the efficiency spirit of the rest of the system.

While working on arithmetic code in MetaPRL, we wanted to create an implementation in which as much as possible could be reused between different logical theories of MetaPRL (both existing ones and any that could be added to MetaPRL in the future). However our main focus in this work is adding arithmetic to MetaPRL's implementation of NuPRL type theory.

Since we want all decision procedures to output explicit proofs of arithmetic inferences, we need to have a complete explicit axiomatization of arithmetic (as opposed to having large parts of the axiomatization in the form of trusted code of decision procedures). In this paper we propose such an axiom system and describe a proof constructing procedure (similar to a version of Arith implemented in Coq [7]) which succeeds in the same cases as NuPRL's Arith, while also generating an actual proof.

2 Choosing the Axioms

The first choice we needed to make is whether to define the arithmetical operators and types as primitive, postulating all the relevant axioms, or to define everything trough existing constructors. For example, we could have attempted to implement the type of natural numbers as `Unit List`. The choice we made is to try to pick a set of axioms that would be universally true across all the reasonable ways of defining the arithmetical primitives. These axioms are added to the system as basic postulates of the type theory; however at a later point we

could derive them from other type constructors (using MetaPRL's derived rules mechanism [12]).

Before defining the set of axioms, one has to choose either integer or natural numbers as a basic type (and later define the remaining type via the chosen type). Both Arith [1] and Sup-Inf [14] use integers as primitive; defining natural numbers on top of integers is more straightforward than the opposite approach. For these reasons we chose to use integers as a primitive type. We used list of axioms from [1] as a prototype.

Another important choice that we had to make is the style of equality reasoning. There are two types of equalities in PRL type theory:

(A) Two terms can be equal as elements of a certain type. For example, two terms $\lambda x.t_1[x]$ and $\lambda x.t_2[x]$ would be equal as elements of a type $A \to B$ (written as "$\lambda x.t_1[x] = \lambda x.t_2[x] \in A \to B$") if for equal inputs of type A, t_1 and t_2 produce equal outputs of type B. Note that two terms could be distinct elements of one type and at the same time be equal in another type — for example, $\lambda x.t_1[x] = \lambda x.t_2[x] \in \texttt{Void} \to T$ is true for arbitrary t_1, t_2 and T.

(B) Finer-grained computational/definitional equality[8] specifies that two terms refer to objects that are not just equal, but are actually *identical*. For example, the terms $\lambda x.a[x]\, b$ and $a[b]$ are computationally equal (written as "$\lambda x.a[x]\, b \equiv a[b]$"); terms $1 + 2$ and 3 are also computationally equivalent.

All things being equal, the equalities of the second kind are easier and more efficient to use. In PRL type theory, we are always allowed to replace a term with a computationally equal one; however we can only replace a term with an equal one when we can prove that the context will tolerate the equalities of the given type. In other words, to be allowed to replace $C[t_1]$ with $C[t_2]$, it is insufficient to be able to prove that $t_1 = t_2 \in T$; we are also required to prove a *well-formedness assumption* stating that C respects the equalities of type T.

Not surprisingly, in our axiomatization we pick computational equality over the equality in a type whenever possible. At the same time we chose to still include the typing assumptions in most of the computational equivalence rules. For example, the commutativity of addition rule is as follows:

$$\frac{\Gamma \vdash a \in \mathbb{Z} \qquad \Gamma \vdash b \in \mathbb{Z}}{\Gamma \vdash (a + b) \;\equiv\; (b + a)} \qquad (add\_Commut)$$

Assumptions look redundant in this rule (at least as long as the + operator is not overloaded). However as we explained above, we want our axioms to stay valid for different formalizations of integers. In particular, in list implementation of natural numbers (with the type of integers being defined as a disjoint union of two list types and addition defined using recursion over lists) the above rule will be provable only in presence of the typing assumptions, as we would need to know that a and b are lists to be able to use their inductive properties.

We certainly want all arithmetic relations ($=, <, >$, etc) to be decidable. In PRL type theory, there are two different ways of defining a decidable relation.

The straightforward approach is to define a predicate (e.g. a function returning a *proposition*) on numbers with an additional axiom stating that this predicate happens to be decidable. The alternative is to postulate an existence of a function on numbers that returns a *boolean* result. To better understand the difference between to choices, it is useful to keep in mind that PRL type theory is *constructive*. A PRL proposition P is identified with a type of all *constructive witnesses* for P; there can be many different propositions and the type of all propositions is a pretty complicated one. By contrast, PRL booleans is a 1-bit type containing just the two boolean constants. While equivalent propositions could be very different, the booleans are completely transparent — if two booleans happen to be equivalent, they must be identical. Because of the latter feature of the boolean type, the rule

$$\frac{\Gamma \vdash a \in \mathbb{Z} \qquad \Gamma \vdash b \in \mathbb{Z} \qquad \Gamma \vdash c \in \mathbb{Z}}{\Gamma \vdash (a < b) \equiv ((a+c) < (b+c))} \qquad (lt\_addMono)$$

will be valid in different implementations of $\mathbb{Z}$, as long as $<$ relation is defined via a boolean comparison function. But if $<$ is defined directly as a proposition, then the best that can be guaranteed is $(a < b) \Leftrightarrow ((a+c) < (b+c))$, which is significantly weaker.

When implementing support for numerals, we could either take the traditional route of building all the numerals using 0 and successor function, or we could simply *expose* MetaPRL's built-in numbers implementation. First approach looks more reliable (and trustworthy) since one only needs to trust the proof checker; however this approach is unbearably slow when one wants to do actual computation using these numerals. Second approach looks less reliable as built-in arithmetic now has to be trusted; however one might argue that it is not adding any new code to the trusted code base, but instead just exposing what is already a part of the prover. In the end, we decided to implement the second approach.

3 Axioms We Chose

In this section we provide an outline of our axiomatization, with an overview of the classes of axioms and some examples. The full list of axioms may be found in the listing of the MetaPRL theories [5, modules Itt_int_base and Itt_int_ext].

(A) Typing properties:

$$\frac{}{\Gamma \vdash \mathbb{Z}\,\mathrm{Type}} \qquad (type\_of\_Int)$$

$$\frac{}{\Gamma \vdash number\{n\} \in \mathbb{Z}} \qquad (type\_of\_number)$$

where `number` is the arithmetical *numeral* operator (e.g. `number`$\{n\}$ stands for an arbitrary numeral constant).

(B) Numbers are computationally transparent — two equal integers will necessarily be identical:

$$\frac{\Gamma \vdash\ a = b\ \in \mathbb{Z}}{\Gamma \vdash\ a \equiv b} \qquad (intCongruence)$$

(C) Reduction of operations (and relations) on integer constants to meta-level operations on integers, e.g.:

$$\frac{}{\Gamma \vdash\ (number\{i\} + number\{j\})\ \equiv\ number\{i +_m j\}} \, (reduce\_add\_meta)$$

where $+_m$ performs addition of numeral constants using underlying internal arithmetic. This rewrite makes possible an evaluation from $1 + 2$ to 3.

(D) Well-formedness of operations and relations. As with any new operations we have to say term of what type it constructs, e.g.:

$$\frac{\Gamma \vdash\ a = a' \in \mathbb{Z} \qquad \Gamma \vdash\ b = b' \in \mathbb{Z}}{\Gamma \vdash\ (a + a') = (b + b') \in \mathbb{Z}} \qquad (add\_wf)$$

(E) Equivalence of propositional and boolean relations. These two rules are actually define $=_b$ for integers via equality in $\mathbb{Z}$:

$$\frac{\Gamma \vdash \uparrow (a =_b b) \qquad \Gamma \vdash\ a \in \mathbb{Z} \qquad \Gamma \vdash\ b \in \mathbb{Z}}{\Gamma \vdash\ a = b \in \mathbb{Z}} \qquad (beq\_int2prop)$$

where $\uparrow (t)\ ::=\ (t = true\ \in\ \mathbb{Z})$.

$$\frac{\Gamma \vdash\ a = b \in \mathbb{Z}}{\Gamma \vdash\ (a =_b b) \equiv true} \qquad (beq\_int\_is\_true)$$

As it was said we decided to define boolean versions of $=$, $<$, etc as primitive and express propositional inequalities using boolean ones. However equality is so fundamental in PRL type theory that we decided to have both boolean and propositional equality as primitives and have rules that constitute their equivalence.

(F) Ring axioms — commutativity, associativity of $+$ and $*$, distributivity, properties of 0 and 1, e.g.:

$$\frac{\Gamma \vdash\ a \in \mathbb{Z}}{\Gamma \vdash\ (a + 0) \equiv a} \qquad (add\_Id)$$

Here type condition on a is necessary because the rule establishes bidirectional equivalence relation and we, of course, do not want to allow replacing arbitrary term a with $a + 0$.

(G) Axioms of $<$-order ($<$ is irreflexive, transitive, asymmetric, discrete), connection between $<$ and arithmetic operations, e.g.:

$$\frac{\Gamma \vdash\ a \in \mathbb{Z} \qquad \Gamma \vdash\ b \in \mathbb{Z}}{\Gamma \vdash\ ((a\ <_b\ b)\ \wedge_b\ (b\ <_b\ a)) \equiv false} \qquad (lt\_Reflex)$$

$$\frac{\Gamma \vdash a \in \mathbb{Z} \qquad \Gamma \vdash b \in \mathbb{Z}}{\Gamma \vdash ((a <_b b) \vee_b (b <_b a) \vee_b (a =_b b)) \equiv true} \quad (lt\_Trichot)$$

$$\frac{\Gamma \vdash a \in \mathbb{Z} \qquad \Gamma \vdash b \in \mathbb{Z}}{\Gamma \vdash (a <_b b) \equiv (((a + 1) =_b b) \vee_b ((a + 1) <_b b))} \quad (lt\_Discret)$$

First rule constitutes irreflexivity of $<$ relation, second constitutes that $<$ is a linear order and the third one constitutes discreteness of integers. These rules define properties of $<_b$ — the boolean version of $<$ relation; propositional version of $<$ and other inequalities are defined via it below, their properties are derivable from properties of $<_b$.

(H) Induction and definition of primitive recursion over $\mathbb{Z}$:

$$\frac{\begin{array}{l}\Gamma;\ n:\ \mathbb{Z};\ \Delta[n];\ m:\ \mathbb{Z};\ v:\ m < 0;\ z:\ C[m+1] \vdash C[m] \\ \Gamma;\ n:\ \mathbb{Z};\ \Delta[n] \vdash C[0] \\ \Gamma;\ n:\ \mathbb{Z};\ \Delta[n];\ m:\ \mathbb{Z};\ v:\ 0 < m;\ z:\ C[m-1] \vdash C[m]\end{array}}{\Gamma;\ n:\ \mathbb{Z};\ \Delta[n] \vdash C[n]} \quad (intElimination)$$

This rule is our formulation of the induction principle. We cover both negative and positive numbers in a single rule, so we have two separate induction steps.

The next rewrite is a part of definition of `ind` — the primitive recursion operation. We have two more (for zero and positive cases) rewrites to define `ind`.

$$\frac{\Gamma \vdash x < 0}{\Gamma \vdash \begin{array}{l}\texttt{ind}\{x; i, j.down[i; j]; base; k, l.up[k; l]\} \equiv \\ \qquad down[x; \texttt{ind}\{(x+1); i, j.down[i; j]; base; k, l.up[k; l]\}]\end{array}} \quad (reduce\_ind\_down)$$

(I) Expression of subtraction via negation, $>, <=, >=$ via $<$, propositional relations via boolean relations (except equality), e.g.:

$$(a - b) ::= (a + (-b))$$

$$(a < b) ::= (\uparrow (a <_b b))$$

$$(a \leq_b b) ::= (\neg_b (b <_b a))$$

(J) Inductive definition of integer division and remainder operations, e.g.:

$$\frac{\Gamma \vdash 0 \leq a \qquad \Gamma \vdash a < b \qquad \Gamma \vdash a \in \mathbb{Z} \qquad \Gamma \vdash b \in \mathbb{Z}}{\Gamma \vdash (a \,\%\, b) \equiv a} \quad (rem\_baseReduce)$$

4 Automation of Arithmetic Reasoning

As we have mentioned in the introduction, NuPRL has two decision procedures (Arith and Sup-Inf) automating the arithmetical reasoning. They both work with

hypotheses and conclusion [5] in the form of quantifier free Presburger formulas [13], which are essentially arithmetic relations among linear forms; all non-linear subterms (after conversion of every polynomial to its canonical form) are considered to be variables of linear forms.

Arith is very limited in its proof-power but it is relatively fast. Depending on the kind of equation it gets as an input, it runs in either polynomial or exponential time (only $\neq$ in hypotheses and $=$ in conclusion add exponential part). Arith proves simple inequalities; specifically, it can prove inequalities that logically follow from hypotheses by associativity and commutativity of addition and multiplication, properties of 0 and 1, reflexivity, transitivity and weak monotonicity of inequalities. Weak monotonicity is the `ge_addMono` rule:

$$\frac{\Gamma \vdash a \in \mathbb{Z} \quad \Gamma \vdash b \in \mathbb{Z} \quad \Gamma \vdash c \in \mathbb{Z}}{\Gamma \vdash (a \ \geq_b \ b) \ \equiv \ ((a+c) \ \geq_b \ (b+c))} \qquad (ge\_addMono)$$

with restriction that c has to be a numeral constant (or be reducible to one).

A big advantage of Arith is that it uses proof by contradiction [6] and constructs contradictory inequality that follows from assumptions (together with negated conclusion). This is what allows us to construct an actual proof from axioms based on Arith algorithm.

Sup-Inf is much more powerful. When used over rational numbers, Sup-Inf is complete and can provide counterexamples in case a proof fails. Sup-Inf has exponential complexity with respect to the number of equations.

As opposed to Arith, Sup-Inf algorithm does not provide a straightforward migration path that would have allowed turning it into a proof-building tactic. However Mayr's initial investigations [10] suggest that it should be possible to achieve this transformation; even if in a less direct manner.

Since Arith tactic implementation is clearly simpler and more straightforward than Sup-Inf, Arith seems to be a better choice for the initial testing of our axiomatization. As a result, we decided to start our work on proof automation with implementation of Arith procedure. Currently we have a working implementation that supports $+$, $-$(both unary and binary),$*$ and $=$,$\neq$,$<$,$>$,$<=$,$>=$ with arbitrary number of nested negation around them; the only unsupported operations are division and remainder.

5 Implementation of Arith

Our implementation provides the user with two main proof procedures — `normalizeC` and `arithT` [7]:

[5] The PRL type theory is formulated in a single-conclusion sequent form.

[6] Since it only involves decidable relations, the proof it generates is still valid in intuitionistic theory.

[7] The **C** and **T** suffixes is the MetaPRL convention for marking the class of a tool. **C** is used for functions that perform term rewriting — we call them **c**onversions and

The `normalizeC` rewriting procedure is used to rewrite polynomials. When applied to a polynomial term, `normalizeC` converts it to its canonical form (e.g. normalizes it). When applied to a term that has polynomial subterms in it, `normalizeC` will normalize all the polynomial subterms of the given term. For example, if a proof assumption has a form of an equality, one can apply `normalizeC` to the whole assumption and it will normalize both sides of the equality.

Example: The canonical form of $((b * 2 * (a + c)) - (a * b)) + 1$ is $1 + (a * b) + (2 * (b * c))$

In traditional decision procedures normalization step will be usually performed based on some representation of arithmetic terms that is internal to the procedure. In our case we perform an in-theory normalization — as we normalize, we prove every step, and eventually we build a proof of the equality between the original polynomial and the normalized one. In-theory normalization has a higher complexity since commutativity and associativity rules normally only allow swapping neighboring subterms. In-theory normalization also means having to work within a pretty restrictive set of allowed transformations; this makes it noticeably trickier than the "unsupervised" normalization with dedicated representation for arithmetic terms.

The canonical form of a polynomial is achieved by the following steps:

(A) Get rid of subtraction.
(B) Open parentheses using distributivity, move parentheses to the right using associativity of addition and multiplication, preform the basic simplifications (such as $0 \cdot a \to 0$, $1 \cdot a \to a$).
(C) In every monomial, sort (using the commutativity axiom) multipliers in increasing order, with numerical constants pushed to the left (We put coefficients first because later we have to reduce similar monomials). Multiply the constants if there is more than one numeral in one monomial; but if monomial does not have a constant multiplier at all, put 1 in front of it for uniformity.
(D) Sort monomials in increasing order, reducing similar monomials on the fly. As in previous step, numerals are pulled to the left (i.e. considered to be the least in the sort order).
(E) Get rid of zeros and ones in the resulting term.

The `arithT` proof search procedure implements Arith [1]:

(A) First it checks whether the conclusion of the goal sequent is an arithmetic fact, and if so, moves it into hypotheses in negated form (using reasoning by contradiction).
(B) Next, `arithT` converts all negative arithmetic facts in hypotheses to positive ones (it adds new hypotheses, and also leaves the original ones intact). Since there may be several nested negations, this step will be applied repetitively.

conversionals; **T** is used for the class functions capable of performing arbitrary proof search — **t**actics and **t**acticals.

(C) After that it converts all positive arithmetic facts in hypotheses into $\geq$-inequalities.
(D) Now every $\geq$-inequality is normalized — we use `normalizeC` to normalize the polynomials on both sides of every inequality.
(E) Then it tries to find the contradictory inequality that logically follows from that normalized $\geq$-inequalities and proves this implication. This problem is reduced to search for positive cycle in a directed graph. If successful, the resulting inequality will be derived from hypotheses.
(F) Finally, false is derived from found inequality, thus completing the proof by contradiction.

6 Ongoing Work and Future Directions

In this work we were able to come up with a very computation–oriented axiomatization of basic arithmetic. The initial experience of being able to use this axiomatization as a basis for creating LCF-style tactics for automating arithmetical reasoning was positive as well. However; we are still in somewhat early stages of this work — a lot more proof automation is needed and we now see several areas where we could improve the existing implementation as well.

In the nearest future we are going to investigate several of these challenges.

We are planning to further evaluate the usefulness and value of computational rewrites in our implementation of Arith— both from performance and proof size viewpoints. Currently, computational rewrites seem to be the right choice for polynomial normalization; however it is not as clear now how convenient they are going to be, for example, when trying to reduce different forms of inequalities to some canonical form.

Current implementation of arithmetical proof automation is neither easily extendable nor flexible. Currently we use a hardcoded set of rules and rewrites, and we would like to replace it with a more flexible declarative code. Two obvious points of improvement are polynomial normalization and reduction of different inequalities to one type. The standard approach for building extendable algorithms in MetaPRL is resources and resource annotation mechanism [11, Section 4.3] that allow splitting complicated proof search procedures into derived rules and short declarative annotations on those rule.

Currently arithmetical reasoning uses only one resource-driven procedure — `reduceC`. In arithmetical theories it is used to perform rewrites that simplify terms (e.g. $a+0 \rightarrow a$), but the `reduceC` is not specific to arithmetic, it is capable of performing a very large variety of simplifications and reductions (such as, for example, β-reduction, $\lambda x.t[x]\, a \rightarrow t[a]$). However, `reduceC` can not be used for transformations like commutativity ($a + b \longleftrightarrow b + a$), since this transformation does not has any clear directional properties.

In general, the rewriting task needs to be better classified and partitioned. Currently we only have two major groups of computational transformations — reductions and simplifications performed by `reduceC` and normalizations performed by `normalizeC`. However it is clear, that in many cases a much more

fine-grain control is needed. In particular, it appears to be necessary to start distinguishing between reductions and simplifications.

In the introduction we mentioned that efficiency is an important goal for the MetaPRL community. We did not yet have a chance of doing any comprehensive performance evaluation of our implementation of Arith, but this is something we are hoping to be able to do in the near future.

And, of course, we are planning to keep adding new proof automation to the system. We are planning to continue our investigation of possible approaches to turning Sup-Inf into a tactic (possibly building on Mayr's work [10]).

References

1. T. Chan. An algorithm for checking PL/CV arithmetic inferences. In G. Goos and J. Hartmanis, editors, *An Introduction to the PL/CV Programming Logic*, volume 135 of *Lecture Notes in Computer Science*, appendix D, pages 227–264. Springer-Verlag, 1982.
2. Robert L. Constable. On the theory of programming logics. In *Proceedings of the 9th Annual ACM Symposium on the Theory of Computing,* Boulder, CO., pages 269–85, May 1977.
3. Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the NuPRL Development System.* Prentice-Hall, NJ, 1986.
4. Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir N. Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu. MetaPRL — A modular logical environment. Accepted to the TPHOLs 2003 Conference, 2003.
5. Jason J. Hickey, Brian Aydemir, Yegor Bryukhov, Alexei Kopylov, Aleksey Nogin, and Xin Yu. A listing of MetaPRL theories. `http://metaprl.org/theories.pdf`.
6. Jason J. Hickey, Aleksey Nogin, Alexei Kopylov, et al. MetaPRL home page. `http://metaprl.org/`.
7. Daniel Hirschkoff. *Nodesat*, an arithmetical tactic for the Coq proof assistant. Technical Report 96-61, CERMICS, Noisy-le-Grand, April 1996.
8. Douglas J. Howe. Equality in lazy computation systems. In *Proceedings of the 4^{th} IEEE Symposium on Logic in Computer Science*, pages 198–203, Asilomar Conference Center, Pacific Grove, California, June 1989. IEEE, IEEE Computer Society Press.
9. Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.
10. S. Tobias Mayr. Generating primitive proofs from SupInf. Communicated to the NuPRL group at Cornell University, 1997.
11. Aleksey Nogin. *Theory and Implementation of an Efficient Tactic-Based Logical Framework.* PhD thesis, Cornell University, Ithaca, NY, August 2002.
12. Aleksey Nogin and Jason Hickey. Sequent schema for derived rules. In Victor A. Carreño, Cézar A. Muñoz, and Sophiène Tahar, editors, *Proceedings of the 15^{th} International Conference on Theorem Proving in Higher Order Logics (TPHOLs*

2002), volume 2410 of *Lecture Notes in Computer Science*, pages 281–297. Springer-Verlag, 2002.
13. M. Presburger. Über de vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchen, die addition als einzige operation hervortritt. In *Comptes Rendus du Premier Congrès des Mathématicienes des Pays Slaves*, pages 92–101. Warsaw, 1927.
14. Robert E. Shostak. On the SUP-INF method for proving Presburger formulas. *Journal of the Association for Computing Machinery*, 24(4):529–543, October 1977.

Part II

Language Embeddings

A Framework for Multicast Protocols in Isabelle/HOL

Tom Ridge and Paul Jackson

School of Informatics, University of Edinburgh
Kings Buildings, Edinburgh EH9 3JZ, UK
T.J.Ridge@sms.ed.ac.uk, pbj@inf.ed.ac.uk

Abstract. Following a recent paper [1], we are mechanising a framework for modelling multicast protocols, and simplifying their verification. The Pragmatic General Multicast protocol (PGM) is a real protocol to which the results apply. The mechanisation is being carried out in Isabelle/HOL.

1 Introduction

The starting point of our work described here was a recent paper by Maidl and Esparza [1] on simplifying the verification of one class of distributed algorithms, namely multicast protocols. The paper [1] considered the problem of establishing properties of a multicast protocol that are true for all configurations of distributed nodes participating in the protocol. An example property is:

> for all receivers r, if a sender s multicasts a message m (identified by some sequence number), then eventually r receives m, or r realises that m has been lost.

It identifies families of protocols for which it is sufficient to check such properties on some restricted class of configurations. For example, in a very simple case, it might be sufficient to check the protocol only for cases where all receivers are directly connected to the sender. The paper [1] doesn't directly propose how the properties on these restricted configurations should be checked, but plausibly conjectures that it ought to be simpler. There easily might still be an infinite number of restricted configurations, so model checking is not always obviously an answer.

The paper [1] contains several pages dense with definitions and the full version has an appendix with 15 pages of closely-argued intricate proofs. The authors of [1] believe the results correct, but state that they still would much appreciate the assurance of having the proofs checked by a theorem prover. This was the initial impetus for beginning the work described here.

In subsequent discussions of the motivations for mechanically formalising this paper, we have identified a number of overlapping concerns which are shaping our approach.

Most immediately the work is a case study in providing mechanical assurance of the correctness of a pen-and-paper formal methods development. This pleases not only the development's authors, but also anyone wishing to make use of the development. Concerns here include the ease and rapidity with which a mechanical development can be realised as well as the reusability of the development. It is inevitable that the formalism and the proof methods will evolve as they are extended to wider and different classes of protocols. A consequence is that we cannot settle for a once and for all tour-de-force rendering of the paper. We must place emphasis on providing automation to shorten the human time needed to produce any one mechanical development. We must also seek good reusable foundational theories.

A more debatable concern is that of readable definitions and proofs. Esparza – one of the authors of [1] – notes in an email to us that he doesn't care much about this. On the other hand, we can argue that some degree of readability of proofs, and closeness of the mechanised definitions to the paper definitions, ought to hasten development reworking. It should also assist the developers with gaining insights into the logic of their arguments and tracking down problems.

From the point of view of distributed algorithm verification, we see this work as an excellent case study for a number of reasons.

1. The focusing on broadcast protocols introduces much structure that streamlines the definition of protocol instances and simplifies the task of identifying and automating common patterns of reasoning.

   The first author in earlier work considered using the IOA formalism for verifying distributed algorithms, since already theorem proving support exists for it [2, 3]. The IOA formalism is much more general than the framework in [1]. However then, to tackle multicast protocols with it, one would need to add on top a whole new layer of structure. This would make for heavy-weight awkward-to-manipulate definitions.
2. The framework aims immediately to be applicable to whole families of protocols, even though it is developed with the PGM protocol [4] as a primary protocol instance of interest. One immediate step we are considering for further work is attempting to deploy the framework on the LRMP broadcast protocol.

As mentioned earlier, the paper [1] doesn't directly propose how to check properties. Another opportunity for further work is to explore whether, if we check properties by theorem proving, the restricted configurations are indeed significantly simpler to work with. Even if not, we anticipate that much of the definitional structure and reasoning machinery we develop will still be of use.

The rest of this paper contains an overview of [1], followed by some details of the mechanisation and discussion of our results so far. We end by comparing our work to other work in the area, and discussing possibilities for future work in more detail.

2 Overview of Framework Introduced by [1]

2.1 Modelling Multicast Protocols

We consider multicast protocols that operate over communication networks with static tree-shaped topologies.

Given a tree representing a network, we assign to the root node a *sender* agent, to each leaf a *receiver agent* and to all other nodes *network element agents*. We assume networks are homogeneous, all network element agents are the same, as are all receiver agents.

Tree edges name bidirectional *channels* between nodes that can hold *messages*. Channels are modelled using multisets rather than queues to allow for arbitrary reordering of messages.

Agents are modelled essentially as non-deterministic Mealy machines. In one step, an agent possibly inputs a message from a channel it is connected to, changes its internal state, and possibly outputs one message upward in the tree towards the root, and another message downward. Agents can be connected to multiple downward channels and they have some control over which children downward messages are inserted into.

A *protocol* is specified by describing sender, receiver and network element agents. A *protocol instance* is a pair of a tree giving a network topology and a protocol. Given a *protocol P*, we denote the set of *protocol instances N(P)*.

The framework in [1] aims to cover families of protocols that are characterised by giving properties agents are required to satisfy. For example, a property might specify what messages an agent can output, given the previous history of messages the agent has input and output. Rather than leave the structure of the state of agent machines unspecified and only constrained by such properties, [1] decided to use a canonical 'most abstract' representation of an agent's current state, namely the previous history of messages the agent has input and output. The state of a protocol instance is then fully given by the abstract state of the agent at each node (in [1] this is called a *network history*). It is not necessary to explicitly include the state of each channel in the protocol instance state because the state of any channel can be reconstructed by considering the message history of the agents at the channel ends.

Appropriate to distributed systems, an asynchronous interleaving model of concurrency is adopted where each step of a protocol instance is a step of a single agent in the instance.

It is assumed that the properties one is interested in establishing of some protocol are properties of infinite linear traces of protocol instance states, and can be specified in linear temporal logic (LTL). The variant of LTL used is restricted to be stuttering invariant – use of the **X** next-time operator is forbidden – and extended to allow outermost universal quantification over receiver names and message sequence numbers. A fairness notion is introduced so traces are ruled out where for example all senders and receivers make only finitely many steps.

As mentioned in the introduction, the interest is in verifying properties true of all instances of a protocol. To this end, the notion of LTL formula validity

involves not only quantification over all traces of some protocol instance, but also quantification over all instances.

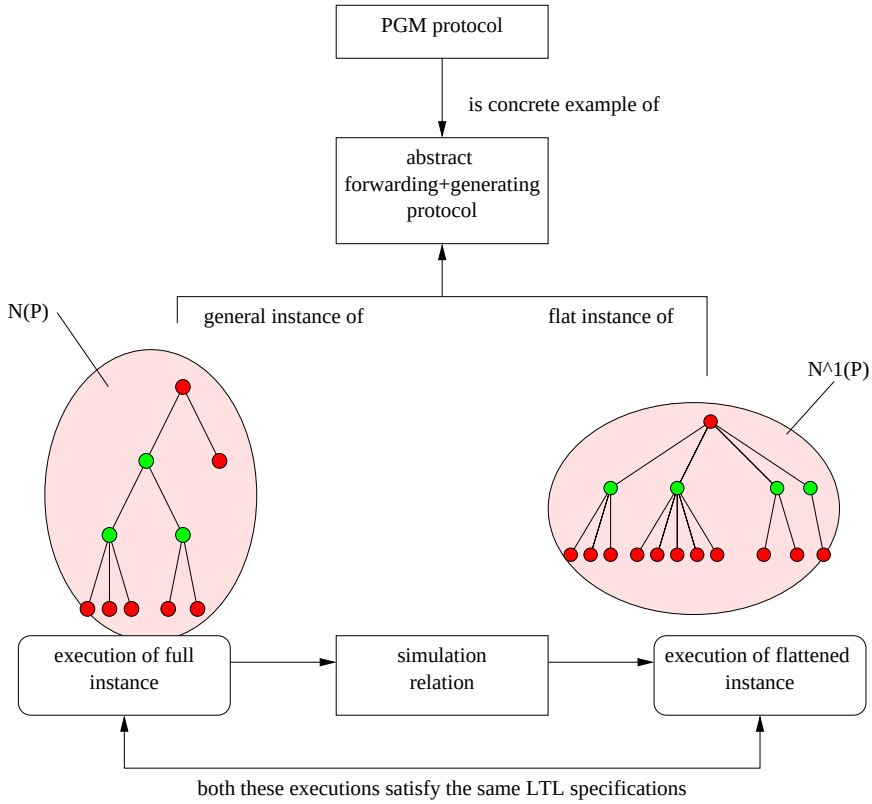
2.2 Verification Simplification

Verifying a property ϕ of a protocol P for every instance T in N(P) may be difficult, and so a common approach is to first reduce the task to checking ϕ for a *restricted class of structures* $N^1(P)$. The main results of the paper [1] isolate sufficient conditions under which such a reduction can take place:

> ...we identify sufficient conditions for a protocol P to be *collapsible*, meaning that a property holds for all instantiations N(P) of P if and only if it holds for the set of instantiations $N^1(P)$ with at most one level of internal elements.

The proof method is to show the existence of a simulation relation between the general instance and a corresponding flattened instance.

Note that we do not verify a particular property for a particular protocol. The existence of a simulation relation shows that to verify a property over all instances of a protocol, it suffices to verify that property over restricted instances (providing the protocol meets certain conditions).



In particular, the conditions are satisfied by network elements that only perform 'forwarding', i.e. messages from sender to receivers are repeatedly passed by a network node to all its children, and messages heading upwards from receivers to sender are similarly forwarded. In addition, the conditions are satisfied by the PGM protocol (an abstract *forwarding+generating protocol*) where network elements have much richer functionality which is used to make communication between the sender and the receivers more reliable and efficient.

2.3 Main Proof Technique: Simulation Relations

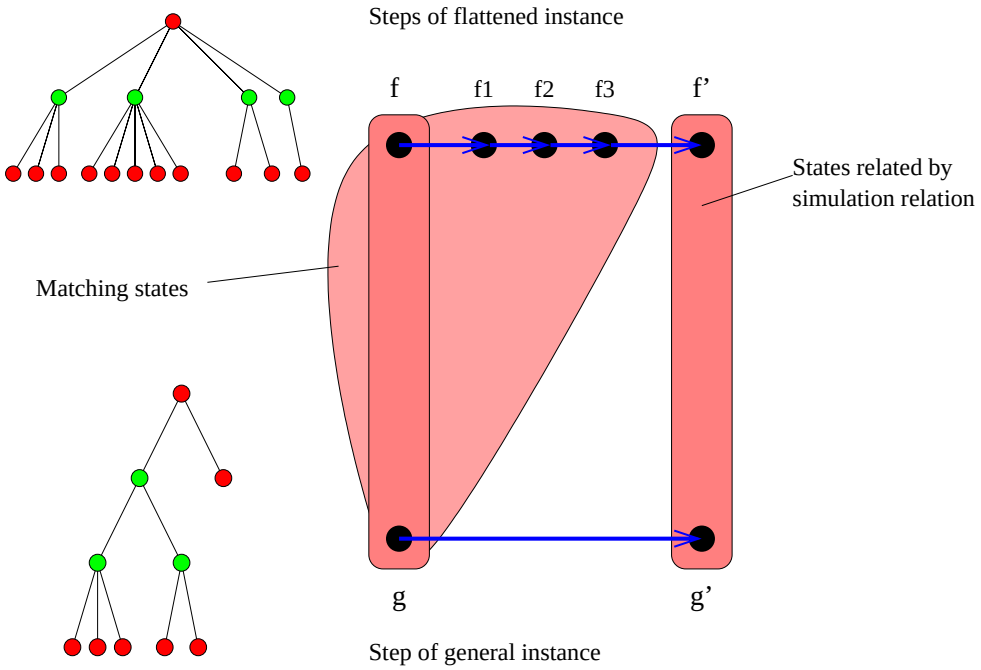
Simulation relations are a proof technique often used when proving properties of distributed systems. In our case, simulation relations ensure that for every execution of the general instance G of a protocol, there is a corresponding execution of the flattened instance F, which satisfies the same LTL formulae.

In general an atomic proposition in an LTL formula could talk about any aspect of state. For instance, we could form the atomic proposition "the third network element is in state s". This might make sense in G, but in the F there might only be one network element, and so such a property cannot be said to make sense for the flat instance. To cope with this, we restrict the atomic properties such that they only talk about the state of the sender and the receivers. This is not such a restriction since most properties (including the example property given earlier) one would want to verify do not mention the intervening network. LTL formulae over such restricted atomic propositions make sense for all general and flat instances.

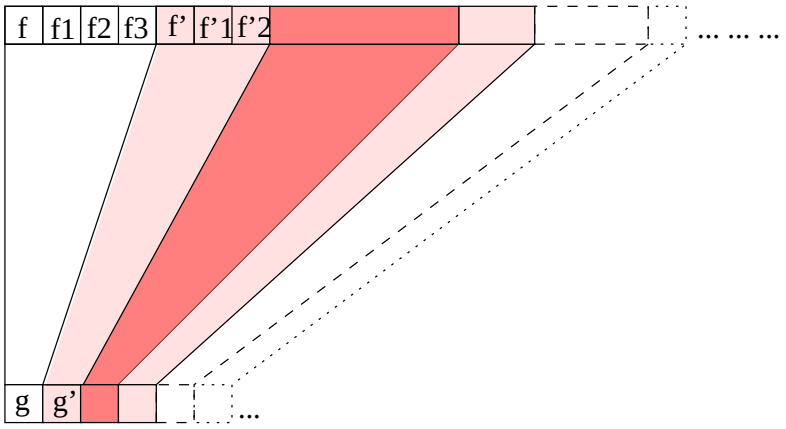
In particular, we can define a relation *Match* between states s of a general instance and states s' of a flattened instance, such that for all atomic propositions p, the general state satisfies p iff the flat state satisfies p.

$$\forall\ p \in ap.\ Match(s,s') \longrightarrow (s \models p) = (s' \models p)$$

A simulation relation is a relation R between states of a general instance G, and states of a flattened instance F, such that, for every transition (g, g') of G, and for all $f \in Rg$ there is a (finite) sequence of transitions (a partial path) of F, say $(f, f1, f2, f3, f')$, such that gRf, $g'Rf'$ and $Match(g, f)$, $Match(g, f1)$, ..., $Match(g, f3)$. Moreover, states related by the simulation relation, also Match.



For a given execution of the general instance, we can piece together successive partial paths to form a corresponding execution of the flattened instance.



Such a path appears to the LTL formula as a stuttered version of the original. Since LTL is stuttering invariant, we have our main result: an execution of a general instance of a protocol P satisfies an LTL formula iff the corresponding execution of the flat instance satisfies that formula. Consequently, if all flat instances satisfy an LTL formula, then so too do all general instances (by contradiction, if there were an execution of the general instance which failed to satisfy

a given LTL formula, then we could form an execution of the flattened execution which likewise failed to satisfy).

2.4 The PGM Protocol

We briefly sketch the PGM protocol [4] to give some idea of the applicability of the current work.

> PGM is a reliable multicast transport protocol for applications that require ordered or unordered, duplicate-free, multicast data delivery from multiple sources to multiple receivers. PGM guarantees that a receiver in the group either receives all data packets from transmissions and repairs, or is able to detect unrecoverable data packet loss [4].

The PGM protocol executes roughly as follows: A sender wishes to multicast a message to a set of receivers, reachable via intermediate nodes in a network having a predetermined (and assumed static) tree topology. The message is tagged with a natural number and passed down to the receivers. If the receivers were to acknowledge (ACK) the receipt of the message, there would be an explosion of ACK messages at the sender. This restricts the extent to which such a protocol could scale. For this reason, receivers keep track of which message they are waiting for, and only signal (negative acknowledge, or NAK) when they detect they are missing a message in the sequence.

The real PGM protocol introduces many additional complications. For instance, to reduce network traffic, network elements keep track of which of their children are missing which messages. Repeated messages are only sent to those that are known to need them. Furthermore, mimicking the behaviour of TCP/IP, a sliding window of messages is maintained by the Sender, who attaches to each message he sends, the earliest message he is prepared to retransmit. Thus, a receiver waiting for a message he missed may find out that the message can never be transmitted anymore. This is probably the origin of the word "pragmatic" in the name of the protocol.

The PGM protocol is a real world, parameterised algorithm which represents a relatively hard target for mechanised verification.

3 Overview of the Mechanisation

The mechanisation breaks down into 3 parts.

- General mathematical knowledge relating to trees and multisets
- General concepts relating to transition systems
  - Kripke structures (simple transition systems)
  - LTL
  - Simulation relations
- Definitions and lemmas relating to the framework of the paper
  - Actions and events

  - Agents
  - Tree networks
  - Protocols
- The results and proofs detailed in the paper.

The mechanisation of trees was relatively straightforward, and most proofs were automatic. The general concepts, being abstract, are the cleanest and most readable part of the formalisation. The definitions used in the framework are interesting in that the notion of agent is very general. The part that deals with the results is a relatively straight forward transcription of the result in the paper: the verification of the existence of a simulation relation.

The actual proofs present in the paper are full of fine grain details, and their mechanised versions are somewhat unilluminating. We prefer to give an overview of the mechanisation itself.

3.1 General Mathematical Knowledge

Trees The protocols execute over communication trees. Many lemmas relating to trees are used implicitly in the proof. Our tree definition is essentially a list of edges, where the first edge attaches to the root (sender) of the tree, and subsequent edges attach to nodes already in the tree.

A tree is either a Sender connected to the environment, or a branch connected to a tree. *Branch* takes a tree, and a channel where the first component of the channel is assumed to be a node already in the tree:

types *'node channel* = *'node* $*$ *'node*

datatype *'a tree* = *Env-Sender* | *Branch 'a tree 'a channel*

There are several advantages to this approach, which we discuss in the conclusion. Briefly, most proofs were automatic, at the expense of an overly concrete representation.

Multisets Multisets are already present in the Isabelle library. To this we added:

- The subset ordering on multisets (the ordering already present in Isabelle is the multiset ordering over a base ordered type).
- Projections of a multiset onto a set of elements.
- Multiplication of a multiset by a natural number.

As noted in the Isabelle library, multisets are in need of a linear arithmetic procedure. This was not available, and so much manipulation of complex terms involving multisets had to be carried out (initially by hand). We discuss further automation of multisets in the conclusion.

3.2 General Concepts of Transition Systems

We formalise transition systems (referred to in the mechanisation as Kripke structures), LTL, several kinds of simulation relation, and various proofs relating them. For example, we formalise that stuttered paths satisfy the same LTL formulae. The proofs involving simulation relations were the most complex.

LTL and Kripke Structures We formalise LTL formulae. We omit the (strong) until operator since it is not necessary to express the sample properties we would like to prove (although our proofs could easily be extended to take this operator into account). We omit the next operator since we seek only to verify formulae which do not include it. Additionally, simulation relations as a method of proof entail an insensitivity to the absolute passage of time.

types *'a path = nat ⇒ 'a*

datatype *'a ltl-formula =*
Init ('a ⇒ bool)
| NOTltl 'a ltl-formula (¬ltl - [90] 90)
| ANDltl 'a ltl-formula 'a ltl-formula (- ∧ltl - [80,80] 80)
| G 'a ltl-formula

Kripke structures consist of a set of start states, and a set of transitions:

types *'s kripke-structure = 's set * ('s transitions)*

Simulation Relations The definition of simulation relations is fairly standard. We are simulating with respect to states, not actions, so instead of requiring that the trace of the partial path matches, we have a condition on the states themselves.

constdefs
is-simulation-relation :: ('a ⇒ bool) set ⇒ ('a,'a) pre-simulation-relation ⇒ 'a kripke-structure ⇒ 'a kripke-structure ⇒ bool
is-simulation-relation ap R KB KA ≡
(∀ b0 ∈ starts KB. ∃ a0 ∈ starts KA. a0 ∈ R b0)
∧ (∀ b b' a. (b,b') ∈ (trans KB) ∧ a ∈ R b ⟶
(∃ app. is-matching-partial-path KA R ap (b,b') a app))

If there exists a simulation relation, then for a given path we can construct a corresponding path that simulates it:

lemma *is-simulated*:
assumes *a*: *aex = corresponding-path R KA ap bex*
and *a1*: *is-simulation-relation ap R KB KA*
and *a2*: *has-path KB bex*
shows *is-simulated-by ap bex aex*

The above is slightly opaque as we have deliberately not included the definition of corresponding path, which is extremely complicated. The idea is simple though. A corresponding path is simply a sequence of partial matching paths pasted together, as in section 2.3.

If bex is simulated by aex, with respect to the atomic propositions of a formula, then that formula cannot distinguish between them:

lemma *pgm-main-theorem*:

assumes *ass*: *is-simulated-by* (*AP f*) *bex aex*
shows *bex* $\models 0$ *f* = *aex* $\models 0$ *f*

3.3 Some definitions and results from [1]

The main results of the paper are to show the existence of a simulation relation between general instances of protocols (meeting certain conditions) and restricted, or flat, instances having only a single layer of network elements. This entails that for every execution of the general instance, there is a corresponding execution for the flat instance. Thus, if all executions of the flat instance satisfy a property ϕ, so too do all executions of the general instance.

To get to this result, many other definitions are made, some of which are represent interesting modelling choices. For instance, the definition of *agents* (the behaviour of a node is termed an agent) is interesting in the way it attempts to be as general as possible.

In this section, we briefly touch on some definitions employed in the mechanisation, and then discuss the mechanisation itself.

Actions and Events types *'msg action* = *'msg*'msg*'msg*

Having divided messages into disjoint sets $M\uparrow$, $M\downarrow$ and $\bot$ (representing "no action"), an action consists of a triple (a,b,c). a represents the message received. On receiving a, b is sent upwards towards the *sender*, whilst c is sent downwards towards *receivers*.

constdefs
is-action :: *'msg action* ⇒ *bool*
is-action ac ≡ *let* (*a*,*b*,*c*) = *ac in a* ∈ *M* ∧ *b* ∈ *insert* ⊥ *M*↑ ∧ *c* ∈ *insert* ⊥ *M*↓

Events occur over trees, whose nodes are named.

types (*'name*, *'msg*) *event* =*'name*'msg**(*'name channel*)**'msg*'msg**(*'name channel set*)

At its simplest, an event occurs at *nad*, some node in a tree. i is received by the node, through channel c. Subsequently *o1* is sent upwards towards the sender, whilst *o2* is sent downwards through C (a subset of the downward channels) towards the receivers. The following definition expresses that the event is well typed. Semantic notions, e.g. the requirement that i actually be in the channel c, come later.

constdefs
is-event :: (*'n*::*env-sender*) *tree* ⇒ (*'n*,*'msg*) *event* ⇒ *bool*
is-event t e ≡ (*let* (*nad*,*i*,*c*,*o1*,*o2*,*C*) = *e in*
is-action (*i*,*o1*,*o2*)
∧ *nad-has-channels t nad c C*
∧ (*nad* = *Sender* ⟶ (*o1* = ⊥ ∧ (*c* = *Ch*↑ *t Sender* ⟶ *i* = ⊥)))
∧ (*nad* ∈ *set* (*res t*) ⟶ *o2* = ⊥)
∧ (*if c* = *Ch*↑ *t nad then* (*i* ∈ *insert* ⊥ *M*↓)
else (*c* ∈ *set* (*Ch*↓ *t nad*)) ∧ (*i* ∈ *insert* ⊥ *M*↑)))

Agents, input output relations and filters An agent is a pair $A = (rho, f)$, consisting of an input output relation and a filter.

types *'msg input-output-rel* =
*('msg node-action list * 'msg) ⇒ ('msg * 'msg) set*
— given a local view of the history of the node, the node decides which messages can be sent upwards or downwards

'msg filter = 'msg node-action list ⇒ 'msg ⇒ bool ⇒ bool
— determines the subset of downward channels through which the message can be sent, given a local history and an indication of whether the message was received through a given channel or not

*'msg agent = 'msg input-output-rel * 'msg filter*

Intuitively an agent A selects the events that can be executed as a function of the past history *nah* (projected onto those actions occurring at the node), and of the current input message *i*. After receiving *i*, the agent selects an event in two steps. First, it non-deterministically selects the messages *o1*, *o2* to be sent upwards and downwards, respectively, as a function of the current trace *nah* and the input message *i*. In the second step, the agent determines the subset of downward channels through which the message *o2* is sent. The agent examines each downward channel $ch \in Ch{\downarrow}\ t\ nad$ in the tree *t* for the given node *nad* and decides whether to send *o2* through it or not depending on the current trace, and whether or not the input message came from that channel. This is a simplified version of a filter as it appears in [1], which suffices for the proof of the first few results. The full definition is employed for proofs relating to forwarding and generating protocols.

constdefs
agent-events :: ('n::env-sender,'msg) tree-network ⇒
('n,'msg) event list ⇒ ('n,'msg) event set
agent-events ta nh ≡ (let (t,a) = ta in {(nad,i,c,o1,o2,C).
let (io-rel, fil) = a nad; nah = (node-action-trace nad nh) in
(o1,o2) ∈ (io-rel (nah,i))
∧ C = { ch. ch ∈ set(Ch↓ t nad) ∧ (ch ≠ c) ∧ (fil nah o2 False = True)}
Un { ch. ch ∈ set(Ch↓ t nad) ∧ ch = c ∧ (fil nah o2 True = True) }
})

types *('n,'msg) transition = ('n,'msg) event list * ('n,'msg) event list*

The states of our system are histories (lists) of events that have occurred (i.e. states are represented implicitly). A transition of the system under an event *e* is represented as a pair of histories, with the second extending the first by the given event.

constdefs
transitions :: (('n::env-sender),'msg) tree-network ⇒ ('n,'msg) transition set
transitions tn ≡ {(h,h'). (is-tn-history tn h')
∧ (∃ e. is-allowable-ne tn h e ∧ h' = e#h) }

3.4 Statement of Main Lemma

So far, we have formalised only the first main result, concerning pure-forwarding protocols. The proof is straightforward, if slightly long-winded: the pen and paper proof took roughly 3 pages, whilst the mechanised version came in at 850 lines. However, this could be significantly reduced with greater automation. Throughout we attempted to keep to the structure of the proof in the paper, although the automation is not currently sufficient to ensure a line for line correspondence. We include the following excerpt from the proof script:

Statement of the main lemma used in the proof of Theorem 3, from [1]. *nh* represents the state of the general instance, which in this case is a network history, that is, a sequence of events that have occurred so far. *nh'* represents the next state, after performing an event (*nad*,*i*,*c*,*o1*,*o2*,*C*). *n-h* represents the corresponding sequence of events that have been performed so far by the flattened instance. The inductive hypothesis gives us that this actually is a valid network history for the flattened instance. *n-h'* represents the subsequent state moved to by the flattened instance (which is an extension of its previous state, as expected). Finally, our inductive hypothesis also give us that *nh* and *n-h'* are related by the simulation relation defined as $\lhd R\ T$. Our goal is to prove that the event taken by the flattened instance is a valid event (*i* is in channel *c*, the event is well formed, etc.), and that the state moved to is related to the state of the general event by the simulation relation.

lemma *pgm-simlemma*: ⟦ *is-tree T*;
nh' = (*nad*,*i*,*c*,*o1*,*o2*,*C*)#*nh*;
(*nh*,*nh'*) ∈ *transitions TA*;
n-h = (*pre-event-mapping T nh*);
is-tn-history TA- n-h;
n-h' = (*event-mapping T nh* (*nad*,*i*,*c*,*o1*,*o2*,*C*))#*n-h*;
(*nh*,*n-h*) ∈ $\lhd R\ T$ ⟧ ⟹
is-allowable-ne TA- n-h (*event-mapping T nh* (*nad*,*i*,*c*,*o1*,*o2*,*C*))
∧ (*nh'*,*n-h'*) ∈ $\lhd R\ T$

4 Conclusions and Related Work

Much has been mechanised, but the paper as a whole has not been fully mechanised. The main result, concerning collapsibility for protocols which are more complicated than pure forwarding protocols, has not been mechanised. We hope to complete the mechanisation in the course of the next few months.

Concerning the goals and motivations listed earlier, most have been largely met. However, one of the main goals has certainly not been met. The work as a whole has taken rather longer to formalise than was initially envisaged. One of the goals of the work was to demonstrate, in the light of concern about the correctness of the proofs in the original paper, that current theorem proving technology is usable by the average practitioner in the area of distributed algorithms, with a minimum of training, to significantly enhance the perceived correctness of their results. This is not the case, although there is reason to hope

that the situation will change. Certainly it is relatively easy to instantiate the results of the paper for different protocols. Moreover, if we desire a similar proof, but concerning a different class of protocol to those considered in the paper, then one hopes that it is relatively easy to change a few definitions and modify the proofs accordingly (Isar format helps with this task). However, for an average practitioner to have carried out the initial mechanisation effort himself, is not realistic. Against this, most of the effort expended was related to the mechanisation of results concerning trees and multisets, results that are implicitly assumed trivial. Since these are part of general mathematical knowledge, one can envisage a time when the libraries and the automation are such that proving these implicit results is no longer required.

In terms of Isabelle features, we utilise axiomatic classes, constants defined at arbitrary type (both discussed below), and Locales. Many of our goals involved extensive restatement of assumptions, and we are currently investigating Locales to see to what extent they can ameliorate the situation. However, our requirements, whilst modest, do not seem to coincide with standard uses of Locales. For instance, we wish to prove results about arbitrary undefined constants, later instantiate those constants in terms of constants in another locale, and prove that they satisfy the assumptions once and for all.

4.1 Formalising Trees

Our formalisation of trees had several advantages, most notably proofs were automatic, or could be easily automated. Against this, our representation is overly concrete (for example, datatype equality does not coincide with equality of the represented trees). Moreover, proofs do not use intuitive notions, or global properties of trees (max acyclic/ min connected say). Presumably results that used these notions would be hard for us to formalise.

We attempted to use Nested Environments (from the Isabelle library), but found that intuitively obvious proofs required considerable care. Furthermore, we only require knowledge of the form of the tree (there is no information attached to the nodes- all that matters is the relation of nodes to other nodes), and so Nested Environments were unsuitable for that reason also.

In terms of Isabelle features, our tree definition utilises axiomatic classes combined with constants defined at arbitrary type. This enables us to talk about the Sender (at a particular type) without having to carry around the related tree. However, this is somewhat non-standard.

We are currently investigating ways to re-mechanise these results. One idea is to follow a general mathematical development of graph theory along the lines of [5], specialised for trees. The definitions and proofs would exist at a more abstract and intuitive level, so the question is to what extent these proofs can be automated. For example, most intuitive proofs would proceed by obtaining a path from sender to a given node, and then arguing that if a property were not true at that node, then some violation of the global tree properties would result. This is a markedly different style to the overwhelming computational/ primitive recursive style employed in the current formalisation.

4.2 Multiset automation

For multisets, we introduce several new constants. If we consider only multiset subset, multiset sum, and multiset difference, then proving an arbitrary proposition from assumptions is equivalent to proving the equivalent proposition over naturals for the count of every member of the multisets concerned. This can be automated relatively easily (although we must unfold the definition of multiset in terms of the counts of its members). However, analogously to dealing with naturals, we may want to have a quick and dirty method for handling simple propositions via. a simpset. We have therefore also lifted the relevant simpset over the naturals to work over multisets (without any explicit unfolding).

When dealing with projections of multisets over different sets, the path is not so clear. Heuristically, we can weaken some assumptions so that they talk about the same projections (the intersection of all sets on which we project), or dually we can strengthen the goals (to remove projections altogether, say). However, it is likely that the right answer involves a subtle interplay of set based reasoning, and linear arithmetic. We intend to consider this problem in more detail in future work.

4.3 Related Work

There is much work related to the formalisation and mechanisation of distributed algorithms. In the Isabelle standard distribution alone, there is HOL/IOA [2], HOLCF/IOA [3], UNITY [6] and TLA [7].

The starting point of this work was HOLCF/IOA. One of the desires was to formalise notions such as simulation relations without the conceptual overhead of Domain theory. This has been achieved, although the formalisation cannot be said to be wholly intuitive. As the work progressed, it was found that natural choices for definitions lead to proofs that seemed somewhat unnatural. HOLCF/IOA had shown that very natural formalisations of results were possible (although with added machinery to handle domain theory), and it was interesting to see how far we could push elementary definitions and techniques. For instance [3] makes the claim that index mappings overly complicate proofs. We agree that they certainly complicate proofs, but that these complications are not insurmountable, and that doing so produces a body of theorems the acceptance of which does not entail the acceptance of the framework of domain theory. However, although the formalisation of simulation relations was sufficient for our work, it is unclear how the formalisation would extend to, say, the filter function on infinite sequences (since we do not explicitly handle non-termination).

Another motivation was to deal with parameterised systems. We believe the IOA formalism [8] may be unsuited to parameterised systems. Many parameterised systems are built out of nodes that have similar behaviour (e.g. all network nodes in the PGM protocol have the same input/output behaviour). Combining similar nodes in IOA is not possible because they are not compatible (their signatures overlap). This in turn is related to the notion of composition in IOA, which does not take network topology into account- rather there is a global

notion of action synchronisation, and topology must presumably be encoded in the action names themselves.

References

1. Esparza, J., Maidl, M.: Simple representative instantiations for multicast protocols. In: TACAS 2003, Springer-Verlag LNCS 2619 (2003) 128–143 Full version at http://www.dcs.ed.ac.uk/home/monika/maidl-pgm-reduction.ps.
2. Nipkow, T., Slind, K.: I/O automata in Isabelle/HOL. In Dybjer, P., Nordström, B., Smith, J., eds.: Proceedings of the International Workshop on Types for Proofs and Programs, Båstad, Sweden, Springer-Verlag LNCS 996 (1994) 101–119
3. Muller, O.: A verification environment for I/O Automata based on formalised metatheory (1998) PhD thesis, TU-Munchen.
4. IETF: RFC3208: PGM reliable transport protocol specification (2001) Available online at http://www.ietf.org/rfc/rfc3208.txt.
5. Bauer, G., Nipkow, T.: The 5 colour theorem in Isabelle/Isar. In Carreño, V., Muñoz, C., Tahar, S., eds.: Theorem Proving in Higher Order Logics. Volume 2410. (2002) 67–82
6. Paulson, L.C.: Mechanizing UNITY in Isabelle. ACM Transactions on Computational Logic **1** (2000) 3–32
7. Merz, S.: Mechanizing TLA in Isabelle. Technical report, (Univ. Maribor, Slovenia) Workshop Verification in New Orientations.
8. Garland, S.J., Lynch, N.A., Vaziri, M.: IOA: A language for specifiying, programming, and validating distributed systems (1997) Available online at: http://citeseer.nj.nec.com/garland97ioa.html.

Verifiable Superposition with PVS

Joni Helin and Pertti Kellomäki

Institute of Software Systems
Tampere University of Technology
Finland

Abstract. We are implementing a new specification language based on superposition, using the logic of the PVS theorem prover as a functional programming language to implement a compiler for the language. The compiler composes superposition steps into specifications.

We have given a semantics for the syntactic forms in terms of temporal logic using the theorem prover. The semantics together with the definition of the compiler allow us to reason about temporal properties of specifications constructed using the compiler.

Since the compiler doubles as a deep embedding of the language, it is also possible to verify properties of the embedded language, such as preservation of invariant properties.

1 Introduction

We are designing and implementing a new specification language called Ocsid [14], which is based on superposition. While the eventual goal is to verify individual specification written in the language, we are also interested in reasoning about properties of the language itself. Some of the properties of the language are of interest when verifying individual specifications, for example preservation of invariant properties in superposition.

We use the PVS theorem prover [17] and a mapping [18] from a subset of the PVS logic to Common Lisp as the implementation vehicle of a compiler for the Ocsid language. The abstract syntax of the language is defined as algebraic data types in PVS, and superposition is defined as a syntactic transformation using recursive functions. We also give a formal semantics for the syntactic constructs in terms of temporal logic, which has been formalized in the PVS logic.

The source code of the compiler doubles as a deep embedding of the Ocsid language in PVS. Let P and Q be predicates, *sup* be the superposition operation, and S and L be a concrete specification and a concrete superposition step respectively. We can use the embedding to verify the following kinds of properties:

$P(S)$	Properties of literal specifications.
$P(sup(S, L))$	Properties of specifications derived using superposition.
$\forall s : P(sup(s, L))$	Properties of literal superposition steps (e.g. all specifications derived using L have some desired property).
$\forall s, l : Q(s, l)$	Properties of the language (e.g. all invariant properties of s hold in $sup(s, l)$).

The rest of the paper is organized as follows. Section 2 describes the background for the research, Sections 3 and 4 outline the implementation of the compiler and the semantics of the Ocsid language. Section 5 discusses verification issues, related work is reviewed in Section 6, and conclusions are drawn in Section 7.

2 Background

In this section we briefly describe superposition as a refinement technique, and review different approaches to embedding languages in logic.

2.1 Superposition

Superposition is an old idea, originally used as a structuring technique for distributed algorithms (e.g. [7, 6, 4]). New functionality, e.g. termination detection is superimposed on a base computation in such a way that the desired properties of the base computation are preserved.

Superposition has been used as a syntactic construct at least in the DisCo [2, 13, 15] and Unity [5] languages. The style of superposition in Ocsid follows the approach taken in DisCo: all assignments to a variable must be introduced in the superposition step introducing the variable. This ensures by construction that invariant properties are preserved in superposition.

2.2 Embedding Languages

There are basically two ways to embed a language L in the logic of a theorem prover: shallow and deep.

In a shallow embedding, syntactic units of L are mapped into syntactic units in the logic. There is no explicit representation of syntax of L in the logic. If the mapping is done using a transformation tool, the correctness of the mapping depends on the tool, which hopefully preserves the relevant part of semantics.

A deep embedding represents syntactic units of L as terms in the logic, and the semantics is given by functions in the logic. Since everything takes place inside the logic, there are no informal links in the chain.

Traditionally, shallow embeddings have been more popular for verification purposes, mainly for two reasons. A shallow embedding avoids one level of indirection caused by the semantic function, so reasoning starts roughly at the level

where one would start when reasoning by hand. Representing the syntax of L in the logic and assigning a formal semantics to it also requires extra work which may not pay off in verification.

When one builds a compiler for a language, one almost inevitably needs to represent the syntax of the source language. Since our compiler is written in the logic of PVS, we can use the same representation for verification purposes, and we get half of the work for a deep embedding for free.

3 The Compiler

In this section we describe the Ocsid compiler. We begin by describing the abstract syntax of specifications and superposition steps and then outline the process of superimposing a step onto a specification.

Syntactic forms are represented using the DATATYPE facility of PVS, which is a concise way of defining algebraic data types. Figure 1 shows the definition of the *specification_syntax* data type. The data type has two constructors *specification* and *invalid_specification* with recognizer predicates *specification?* and *invalid_specification?* respectively. The first constructor takes four arguments: a list of classes, a list of initial conditions, a list of conditional invariants, and a list of actions.

```
specification_syntax: DATATYPE
 BEGIN
  IMPORTING action_syntax, class_syntax, conditional_invariant_syntax

  specification(classes: list[class_syntax],
                initial_conditions: list[expr_syntax],
                conditional_invariants: list[conditional_invariant_syntax],
                actions: list[action_syntax]): specification?
  invalid_specification: invalid_specification?
 END specification_syntax
```

Fig. 1. The algebraic data type representing specifications

Figure 2 shows the data type representing superposition steps. The attributes *required_classes* and *required_actions* list the classes and actions that are to be present in a specification in order for a step to be applicable to a specification. The remaining attributes give the new structure to be superimposed on the specification.

The rest of the data types representing syntax are similar, so we omit them here.

Superposition is defined as a syntactic transformation. If a step is applicable to a base specification, a new specification is constructed from the base

```
superposition_step_syntax: DATATYPE
 BEGIN
  IMPORTING action_extension_syntax, class_extension_syntax

  superposition_step(required_classes: list[class_syntax],
                     required_actions: list[action_syntax],
                     class_derivations: list[derivation_pair_syntax],
                     class_extensions: list[class_extension_syntax],
                     initial_conditions: list[expr_syntax],
                     invariants: list[expr_syntax],
                     action_derivations: list[derivation_pair_syntax],
                     action_extensions: list[action_extension_syntax])
                   : superposition_step?
 END superposition_step_syntax
```

Fig. 2. The algebraic data type representing superposition steps

specification and the step, otherwise the result of superposition is the invalid specification. Figure 3 gives the definition of the *superimpose* function.

```
superimpose(step: superposition_step_syntax,
            spec: specification_syntax): specification_syntax =
  IF applicable?(step, spec)
  THEN specification(extend_classes(class_extensions(step),
                                    derive_classes(class_derivations(step),
                                                   classes(spec))),
                     append(initial_conditions(step), initial_conditions(spec)),
                     cons(make_conditional_invariants(step, spec),
                          conditional_invariants(spec)),
                     extend_actions(action_extensions(step),
                                    derive_actions(action_derivations(step),
                                                   actions(spec))))
  ELSE invalid_specification
  ENDIF
```

Fig. 3. Definition of superposition

Functions *derive_classes* and *derive_actions* introduce new derived classes and actions respectively, and *extend_classes* and *extend_actions* introduce new structure into the classes and actions.

Processing initial conditions is straightforward. Initial conditions of the resulting specification are simply a union of those of the original specification and of the step.

Checking the applicability of a step on a specification requires showing logical implication between actions, which cannot be done automatically in the general case. The compiler checks that the base specification has the required actions with the required roles in order to avoid dangling references to identifiers. Invariants of the step are included as conditional invariants in the result of superposition, where the conditions are action implications. The meaning of a conditional invariant is that if the conditions are true, then the invariant holds.

A derived class or action can have multiple bases. For classes, this is simply multiple inheritance where the result gets all the attributes of the bases. For actions, such derivation corresponds to sequential composition.

The intended semantics of sequential composition is that the state reached by executing the composed action is the same as the one reached by executing the component actions in succession. In the compiler, composition is a syntactic operation that composes a new action from a list of existing actions.

The *compose* function performs sequential composition of two syntactic actions. The roles and parameters of the composite action are simply unions of the respective components, but the guard and body of the new action cannot be constructed in the same way. Whenever expressions in the guard or the body of the second action reference targets of assignments of the first action, these expressions are substituted by the corresponding right hand side of the assignment. The substitution makes the latter part of the composed action behave as though it was executed in a state resulting from the execution of the first action.

Figure 4 shows a fragment of the definition of sequential composition. The functions recurse on lists, and the termination of recursion is guaranteed by the MEASURE clause. Type checking the declarations results in proof obligations to show that the expressions given in the clauses decrease with each recursive invocation.

```
substitute_assignments(conjunct: expr_syntax,
                       assignments: list[assignment_syntax])
                     : RECURSIVE expr_syntax =
  IF null?(assignments) THEN conjunct
  ELSE substitute_assignments(substitute(conjunct, car(assignments)),
                              cdr(assignments))
  ENDIF
   MEASURE length(assignments)

compose(g1, g2: list[expr_syntax], b1: list[assignment_syntax])
     : RECURSIVE list[expr_syntax] =
  IF null?(g2) THEN g1
  ELSE compose(append(g1, (: substitute_assignments(car(g2), b1) :)),
               cdr(g2), b1)
  ENDIF
   MEASURE length(g2)
```

Fig. 4. Fragment of the definition of sequential composition

As expressions are instances of a recursive algebraic data type, the *substitute* function used in *substitute_assignments* follows the structure of the expression by a case analysis, and recursively descends into leaf expressions referencing variables. If a reference mentions a variable other than the assignment's target, an equivalent expression tree is constructed bottom-up on return from the recursion. Otherwise, the constructed expression tree differs at that particular leaf.

One issue to consider when composing action bodies are assignments that target same variables. In this case, an assignment in the latter part of the composed action makes the corresponding earlier assignment extraneous and thus subject to removal.

3.1 An Example

To make the preceding discussion more concrete, we present a small example of how the compiler is used. Figure 5 gives an example specification in the abstract syntax. The "(:" and ":)" brackets are PVS syntax for list literals. Constructors of abstract data types representing Ocsid syntax are indicated with a sans serif font.

```
spec: specification_syntax =
    specification((: class("c", (: "c" :),
                         (: field_declaration("i", "int") :)) :),
                  (: forall_q( formal_parameter("x", "c"),
                               gt( svar_ref("x", "i"), intconst(0))) :),
                  (: :),
                  (: action("a", (: "a" :),
                            (: formal_parameter("p", "c") :), (: :),
                            (: gt( svar_ref("p", "i"), intconst(5)) :),
                            (: assign( svar_ref("p", "i"),
                                       plus( svar_ref("p", "i"),
                                             intconst(10))) :)) :))
```

Fig. 5. An example specification

We superimpose on *spec* a superposition step that augments class c with a new variable, and action a with an assignment to the variable. The superposition step is given in Figure 6.

With these definitions in a theory that imports the definition of the compiler, we can evaluate the superposition of *step* on *spec* by entering an interactive session with the PVS ground evaluator as follows.

```
PVS Ground Evaluation.
Enter a ground expression in quotes at the <GndEval> prompt.
```

```
step: superposition_step_syntax =
     superposition_step(
          (: class("c", (: "c" :), (: field_declaration("i", "int") :)) :),
          (: action("a", (: "a" :),
                        (: formal_parameter("p", "c") :), (: :),
                        (: gt( svar_ref("p", "i"), intconst(5)) :),
                        (: assign( svar_ref("p", "i"),
                                        plus( svar_ref("p", "i"),
                                              intconst(10))) :)) :),
          (: :),
          (: class_extension("c", (: field_declaration("k", "int") :)) :),
          (: forall_q( formal_parameter("p", "c"),
                    gt( svar_ref("p", "k"), intconst(0))) :),
          (: forall_q( formal_parameter("p", "c"),
                    gt( svar_ref("p", "k"), intconst(0))) :),
          (: :),
          (: action_extension("a", (: :), (: :), (: :),
                                  (: assign( svar_ref("p", "k"),
                                          plus( svar_ref("p", "i"),
                                              intconst (1))) :)) :))
```

Fig. 6. An example superposition step

```
<GndEval> "superimpose(step, spec)"
==>
specification(
  (: class("c", (: "c" :), (: field_declaration("k", "int"),
                              field_declaration("i", "int") :)) :),
  (: forall_q(formal_parameter("p", "c"),
              gt(svar_ref("p", "k"), intconst(0))),
     forall_q(formal_parameter("x", "c"),
              gt(svar_ref("x", "i"), intconst(0))) :),
  (: conditional_invariants(
        (: :),
        (: action_refinement_pair(
              action("a", (: "a" :),
                     (: formal_parameter("p", "c") :), (: :),
                     (: gt(svar_ref("p", "i"), intconst(5)) :),
                     (: assign(svar_ref("p", "i"),
                               arbitrary_int_value) :)),
              (: class("c", (: "c" :),
                       (: field_declaration("i", "int") :)) :),
              action("a", (: "a" :),
                     (: formal_parameter("p", "c") :), (: :),
                     (: gt(svar_ref("p", "i"), intconst(5)) :),
                     (: assign(svar_ref("p", "i"),
                               plus(svar_ref("p", "i"),
                                    intconst(10))) :)),
```

```
            (: class("c", (: "c" :),
                     (: field_declaration("i", "int") :)) :)) :),
       (: forall_q(formal_parameter("p", "c"),
                   gt(svar_ref("p", "k"), intconst(0))) :)) :),
 (: action("a", (: "a" :), (: formal_parameter("p", "c") :), (: :),
          (: gt(svar_ref("p", "i"), intconst(5)) :),
          (: assign(svar_ref("p", "k"),
                    plus(svar_ref("p", "i"), intconst(1))),
             assign(svar_ref("p", "i"),
                    plus(svar_ref("p", "i"), intconst(10))) :)) :))
```

4 Semantics

The previous section described the syntactic transformations performed by the Ocsid compiler. In this section we outline the semantics of Ocsid specifications.

In Ocsid, the global state of a system is composed of the values of state variables, which are contained in objects. Similarly to Lamport's Temporal Logic of Actions [16], the values of state variables are drawn from a set *values*, which is a union of all the desired values.

In the current formulation, the possible values are integers, booleans and object references. The following algebraic data type defines the desired tagged union.[1] The data type is parameterized with a type representing references of objects.

```
value: DATATYPE
 BEGIN
  intval(v: int): intval?
  boolval(v: bool): boolval?
 END value
```

In order to give the semantics of Ocsid specifications, we formalize the basic concepts of linear time temporal logic. *State* is an uninterpreted data type, a *behavior* is a sequence of states, and a *variable* is a function from states to values:

```
  state: TYPE+
  behavior: TYPE = sequence[state]
  variable: TYPE = [state → value]
```

Objects are containers of variables, represented with an uninterpreted data type. The mapping between objects and variables is done using *fields*, which are functions from objects to variables. We also need function types for mapping identifiers in the abstract syntax into objects and fields at the semantic level. These are formalized in PVS as

[1] PVS 3.1 allows for untagged unions, but we have not investigated their use yet.

```
object: TYPE+
field: TYPE = [object → variable]
object_naming: TYPE = [id → object]
field_naming: TYPE = [id → field]
```

The mapping between objects and variables does not change during the execution of a system, so can we avoid some parameter passing and declare the mapping as

```
the_field_naming: field_naming
```

The values of state variables are of type *value*, but when doing actual verification, we do not want to deal with the tagged union *value* any more than necessary. We use function overloading and declare three versions of a function returning the value of a variable as a value of the *value* data type, as an integer and as a boolean, respectively:

```
svar_ref(obj: object, field: id, s: state): value =
    the_field_naming(field)(obj)(s)

svar_ref(obj: object, field: id, s: state): int =
    CASES the_field_naming(field)(obj)(s) OF
      intval(i): i
      ELSE arbitrary_int(obj, field, s)
      ENDCASES

svar_ref(obj: object, field: id, s: state): bool =
    CASES the_field_naming(field)(obj)(s) OF
      boolval(b): b
      ELSE arbitrary_bool(obj, field, s)
      ENDCASES
```

PVS automatically chooses the definition to use according to context. For example, if a reference to a state variable appears as an operand to the "+" operator, the definition returning an integer is used.

The semantic function (or rather a collection of overloaded functions) *sem* gives a meaning for the various syntactic forms. As with variable references, we give separate definitions for the integer and boolean semantics of a syntactic form.

Figure 7 gives a definition of the boolean semantics of expressions. Each of the branches of the CASES expression is a pattern that matches a constructor in the DATATYPE definition of *expr_syntax*. The constructors are again indicated with a sans serif font.

Termination of recursion is guaranteed by the "MEASURE e BY ≪" clause, which states that the size of parameter e decreases in each recursive call, when measured with the well-founded measure ≪ (automatically generated by PVS).

```
sem(e: expr_syntax, env: environment, s: state): RECURSIVE bool =
  CASES e
    OF
        boolconst(b): b,
        svar_ref(obj, f): svar_ref(object_naming(env)(obj), f, s),
        gt(e1, e2): sem(e1, env, s) > sem(e2, env, s),
        lt(e1, e2): sem(e1, env, s) < sem(e2, env, s),
        and(e1,e2): sem(e1, env, s) ∧ sem(e2, env, s),
        or(e1, e2): sem(e1, env, s) ∨ sem(e2, env, s),
        implies(e1, e2): sem(e1, env, s) ⊃ sem(e2, env, s),
        not(e1): ¬ sem(e1, env, s),
        equal(e1, e2): sem(e1, env, s):: int = sem(e2, env, s),
        forall_q(q_var, e):
         ∀ (ob: object):
           sem(e,
               env
                 WITH [object_naming :=
                         object_naming(env)
                             WITH [(name(q_var)) := ob],
                       object_typing :=
                         object_typing(env)
                             WITH [(name(q_var)) := of_type(q_var)]],
               s),
        exists_q(q_var, e):
         ∃ (ob: object):
           sem(e,
               env
                 WITH [object_naming :=
                         object_naming(env)
                             WITH [(name(q_var)) := ob],
                       object_typing :=
                         object_typing(env)
                             WITH [(name(q_var)) := of_type(q_var)]],
               s)
    ELSE arbitrary_bool(e, env, s)
    ENDCASES
  MEASURE e BY ≪
```

Fig. 7. The semantics of syntactic forms as booleans

Type checking the definition generates a proof obligation to show that this is indeed the case.

The WITH syntax is function and record overriding. The value of f WITH [e := v] is a function that is otherwise equivalent to f, but returns v for argument e. Overriding of record fields is done similarly.

The CASES in Figure 7 branches for forall_q and exists_q show how the environment in which expressions are evaluated is augmented. The semantic definition quantifies over Ocsid objects, and the environment binds the quantified variable to the syntactic identifier.

We omit the rest of the semantics, and show only how the semantics of Ocsid specifications is defined. A specification is a predicate on behaviors, which is true if the behavior satisfies the initial conditions of the specification, and each step of the behavior is either an execution of one of the actions of the specification, or a stuttering step that does not change any of the variables of the specification. This is expressed formally as

```
sem(s: specification_syntax): [behavior → bool] =
    λ (b: behavior):
      every((λ (e: expr_syntax): sem(e, b)), initial_conditions(s)) ∧
       (∀ (n: nat):
          (∃ (a: action_syntax):
             member(a, actions(s))
             ∧ sem(a, b(n), b(n + 1), classes(s)))
           ∨
           sem(the_stuttering_action, b(n), b(n + 1), classes(s)))
```

5 Verification

Our primary interest is to be able to verify specifications written in the Ocsid language. This mostly involves reasoning about actions at the semantic level, but syntactic reasoning is occasionally useful as well.

We are also interested in verifying that the compiler works as expected. Since the compiler doubles as the formal specification of the language, the compiler is correct by definition (or rather the language is by definition the one implemented by the compiler). However, there are still relevant properties to be verified, for example that the syntactic operation we call sequential composition in fact produces a new action which is semantically equivalent to executing the component actions in succession.

5.1 Verification of Specifications

Specifications in Ocsid are constructed by successive application of superposition steps, starting with the empty specification. Since superposition is defined constructively in terms of recursive functions, deriving the resulting specification is a simple matter of repeatedly rewriting with the function definitions until a specification is obtained.

Verification of invariant properties of a specification is done using the semantics given in Section 4 and standard invariant reasoning.

5.2 Verification of Superposition Steps

More interesting than verification of individual specifications is verification of properties of the form

$$\forall(s : specification) : applicable(L, s) \Rightarrow P(superimpose(L, s)) \qquad (1)$$

where L is a superposition step, *applicable* is a predicate which is true if s satisfies the assumptions made by L, and P is an invariant property.

Superposition steps are a more attractive unit of verification because of the universal quantification over specifications in (1). Verification of a superposition step is amortized over uses of the step. In order to establish $P(superimpose(L, s))$ for a particular specification s, one only needs to establish that $applicable(L, s)$ is true.

5.3 Verification of the Compiler

Verification of the compiler is work in progress.

Since the compiler serves at the same time as a formal definition of the language, there is no need to establish that the compiler is true to the semantics of the language. Instead, we verify that the language implemented by the compiler has certain desirable properties, most prominently preservation of invariant properties by superposition. We also verify that some of the syntactic transformations adhere to the informal interpretation given to them.

Preservation of temporal safety properties by construction is the corner stone of the style of specification developed in the DisCo project and a specification language by the same name [2, 13, 15]. In Ocsid, some temporal safety properties may be violated by superposition, but invariants expressed in terms of state variables are preserved.

Preservation of invariant properties is expressed by the following theorem in PVS (lifts and lifte lift expressions and specifications to temporal syntax).

Theorem 1 (Invariance preservation).

```
∀ (spec: specification_syntax,
   step: superposition_step_syntax, inv: expr_syntax):
  ∀ (b: behavior):
    sem( implies( lifts(spec), box( lifte(inv))))(b) ⇒
     sem( implies( lifts(superimpose(step, spec)), box( lifte(inv))))(b)
```

Verifying Theorem 1 gives a formal basis for incremental development and verification of specifications. Invariant properties that hold in intermediate specifications are guaranteed to hold in the final result.

Another important property is the correctness of sequential composition of actions. The semantic correctness of the syntactical operation *compose* is captured as the following theorem.

Theorem 2 (Composition correctness).

```
∀ (s1, s2, s3: state,
   a1, a2: action_syntax, cl: list[class_syntax], env: environment):
  sem(a1, s1, s2, cl, env) ∧ sem(a2, s2, s3, cl, env) ⇔
   sem(compose(a1, a2), s1, s3, cl, env)
```

6 Related Work

Since correctness of compilers is a rather obvious concern, there is a wealth of work on compiler verification. Perhaps closest to our work is the work on compiler verification using PVS [8, 1, 19], and the work using the ACL2 prover as a programming vehicle for compilers [10].

Even though we are concerned about the correctness of our compiler, the problem formulation is a bit different from compiler verification for programming languages. A compiler for a programming language takes as input a program P_S in the source language and produces another program P_T in the target language. The compiler verifier must then show that the semantics of P_S and P_T are the same according to some suitable semantics.

In our case, P_S is a collection of superposition steps, and P_T is a specification. The semantics we are interested in is given in terms of behaviors, but the only way to give a behavioral semantics to a collection of superposition steps is to compose the steps into a specification.

Since the composition is formally defined by the compiler, the compiler is trivially correct and there is nothing to verify. What we can do is to verify that the language implemented has some desired properties, such as preservation of invariants as explained in Section 5.

The compiler is work in progress towards a language with a template system that allows us to express and verify temporal properties at a more abstract level. Since the process of using templates is relatively complex (instantiating a template produces a superposition step, which is then applied to a specification to produce a new specification), we wish to get the formal details right. Implementing the compiler in PVS gives us the possibility to do formally assisted language design.

As a deep embedding, our compiler is similar to other language embeddings, e.g. Unity in HOL [20], the functional language PCF in HOL [9], the imperative language Sunrise in HOL [11] and Java in Isabelle and PVS [12] to name a few.

The Common Lisp mapping of PVS logic requires us to use a constructive style where recursive functions are used instead of quantifications. For example, instead of writing

$$\forall e : e \in l \Rightarrow P(e)$$

to express that all elements of l must satisfy predicate P, we have to write

$$every((\lambda e : P(e)), l)$$

where *every* is a recursively defined function.

A nice discussion of issues regarding embedding other formalisms in theorem provers can be found in [3].

7 Conclusions

We have presented how a compiler for a specification language is implemented using the logic of the PVS theorem prover. The compiler is a collection of PVS theories, which are mapped to Common Lisp for execution.

The compiler doubles as a deep embedding of the language in the logic of PVS, which gives us the ability to verify a wide range of properties, from invariant properties of specifications to properties of the language itself.

We are using the system for formally assisted language design. The current compiler is work in progress towards a language with a template mechanism for expressing abstract collective behavior. Working within a mechanized logic helps us to get the formal details of the language right.

Acknowledgments

This research was partly supported by Academy of Finland, grants number 100005 and 102536, Kellomäki was funded as a Research Fellow by the Academy of Finland.

John Rushby suggested using the PVS logic as an implementation language.

References

1. The Verifix project home page. At `http://www.info.uni-karlsruhe.de/~verifix/`.
2. The DisCo project WWW page. At `http://disco.cs.tut.fi` on the World Wide Web, 2001.
3. A. Azurat and I.S.W.B. Prasetya. A survey on embedding programming logics in a theorem prover. Technical Report UU-CS-2002-007, University of Utrech, Netherlands, 2002.
4. K. M. Chandy and L. Lamport. Distributed snapshots: determining the global state of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.
5. K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
6. K. Mani Chandy, Jayadev Misra, and Laura M. Haas. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1(2):144–156, May 1983.
7. Edsger W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.
8. Axel Dold, Thilo Gaul, and Wolf Zimmermann. Mechanized verification of compiler backends. In Tiziana Margaria and Bernhard Steffen, editors, *Proceedings of the International Workshop on Software Tools for Technology Transfer STTT'98*, pages 13–22, Aalborg, Denmark, 1998.

9. G. Collins. A proof tool reasoning about functional programs. In J. Von Wright, J. Grundy, and J. Harrison, editors, *Ninth international Conference on Theorem Proving in Higher Order Logics TPHOLs*, volume 1125, pages 109–124, Turku, Finland, 1996. Springer Verlag.
10. Wolfgang Goerigk. *Computer-Aided Reasoning: ACL2 Case Studies*, chapter Compiler Verification Revisited. Kluwer Academic Publishers, 2000.
11. P. V. Homeier and D. F. Martin. Mechanical verification of total correctness through diversion verification conditions. *Lecture Notes in Computer Science*, 1479, 1998.
12. Marieke Huisman. *Reasoning about Java Programs in higher order logic with PVS and Isabelle*. Ipa dissertation series, 2001-03, University of Nijmegen, Holland, February 2001.
13. H.-M. Järvinen, R. Kurki-Suonio, M. Sakkinen, and K. Systä. Object-oriented specification of reactive systems. In *Proceedings of the 12th International Conference on Software Engineering*, pages 63–71. IEEE Computer Society Press, 1990.
14. Pertti Kellomäki. A structural embedding of Ocsid in PVS. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics, TPHOLS2001*, number 2152 in Lecture Notes in Computer Science, pages 281–296. Springer Verlag, 2001.
15. Reino Kurki-Suonio. Fundamentals of object-oriented specification and modeling of collective behaviors. In H. Kilov and W. Harvey, editors, *Object-Oriented Behavioral Specifications*, pages 101–120. Kluwer Academic Publishers, 1996.
16. Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
17. Sam Owre, John Rushby, Natrajan Shankar, and Friedrich von Henke. Formal verification of fault-tolerant architecures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
18. N. Shankar. Efficiently executing PVS. Project report, Computer Science Laboratory, SRI International, Menlo Park, CA, November 1999. Available at `http://www.csl.sri.com/shankar/PVSeval.ps.gz`.
19. David W. J. Stringer-Calvert. *Mechanical Verification of Compiler Correctness*. PhD thesis, University of York, Department of Computer Science, York, England, March 1998. Available at `http://www.csl.sri.com/~dave_sc/papers/thesis.html`.
20. T. E. J. Vos. *UNITY in Diversity: A Stratified Approach to the Verification of Distributed Algorithms*. PhD thesis, Utrecht University, 2000.

Modal Linear Logic in Higher Order Logic
An experiment with COQ

Mehrnoosh Sadrzadeh
msadr016@uottawa.ca *

Abstract

The sequent calculus of classical modal linear logic $KDT4_{lin}$ is coded in the higher order logic using the proof assistant *COQ*. The encoding has been done using two-level meta reasoning in *Coq*. $KDT4_{lin}$ has been encoded as an object logic by inductively defining the set of modal linear logic formulas, the sequent relation on lists of these formulas, and some lemmas to work with lists.This modal linear logic has been argued to be a good candidate for epistemic applications. As examples some epistemic problems have been coded and proven in our encoding in *Coq*: :the problem of logical omniscience and an epistemic puzzle: 'King, three wise men and five hats'

1 Introduction

In this paper we present an encoding of modal linear logic using the Coq proof assistant. The logic has been developed in [8] by adding $KDT4$ type modalities to classical linear logic [5].It has the special feature of adding modalities to linear logic on top of linear exponentials. This encoding allows us to state and prove theorems of that logic using facilities of Coq. Previous work in encoding intuitionistic linear logic has been done by associating the constructs of Coq together with linear logic proofs [11]. In our encoding we are treating the modal linear logic as an *object logic* and Coq's Calculus of Inductive Constructs (CIC) as the *meta logic*. This methodology and its benefits has been presented in [4]. Following this methodology linear logic formulas and sequent rules have been inductively defined using the set of inductive datatypes of Coq.Modal logic, too, has been previously encoded in Coq following the same approach [7] . Our encoding will also take advantage of Coq's inductive constructs for implementing sequent rules. The special feature of our logic is that first we are encoding a modal linear logic and second our sequents are classical in the sense that we are not limiting ourselves to sequents with single formulas on the right hand side.The encoding has been done in three steps: (i) defining

*Ph.D. Student, Department of Philosophy, University of Ottawa, Ottawa, Ontario, K1N 6N5, CANADA

modal linear logic formulas inductively, (ii) defining modal linear logic sequent rules inductively, and (iii) extending the modality to an indexed modality thus making our encoded logic a multi-modal linear logic. Following the work of Hintikka [6], modal logics have been widely used to reason about the knowledge of agents. It has been argued that modal linear logic is a good candidate for a non-idealized epistemic logic [3]. In order to study the capabilities of this logic in dealing with epistemic applications, we will use our encoding to prove two standard problems of epistemic logic.

In what follows we first discuss our motivations in working with this modal linear logic in Coq. Then we give a brief overview of the modal linear logic we use.Some familiarity with both linear and modal logics is assumed. We then present our encoding of modal linear logic, and give examples of its application to two well-known epistemic problems: the problem of logical omniscience, and the "King, three wise men, and five hats" puzzle.Finally, we will discuss briefly why we chose classical over intuitionistic logic for the epistemic application.

2 Motivation

Reasoning about knowledge has been a central issue in epistemology since Plato defined knowledge as *justified belief*. In the twentieth century, the discussion was renewed by the use of formal logic and modal operators to account for propositional attitudes such as *I know that...*or *I believe that....* Thus one could use the tools of modern logic to study how agents reason about knowledge. This new branch of logic, called epistemic logic, has found applications in computer science and economics since its inception in the 1960s. But this epistemic logic has had to deal with a serious drawback known as the problem of *logical omniscience*: if an agent knows that p and knows that 'p implies q', deductive closure requires that the agent also knows that q. This is obviously not the case for real agents: we do not know all the consequences, for example, of the axioms of elementary arithmetic. Nor would it be true of a computer because the resources necessary for the knowledge of q might not be available to it, if for example calculation involves exponential complexity.

Thus, epistemic logic embodies strong idealizations about the deductive powers of the agents. One way to go about solving this problem is to find what are the causes of these idealizations. Following Girard's analysis leading to Linear Logic [5], logical systems are made of logical rules concerning the connectives and structural rules, some of them, known as *contraction* and *weakening* are to blame for idealizations. These rules are shown below:

$$\textit{Contraction} \quad \frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta}\ Left \qquad \frac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta}\ Right$$

$$\textit{Weakening} \quad \frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}\ Left \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta}\ Right$$

Contraction enables us to use a formula infinitely many times in a proof. *Weakening*, on the other hand, allows us to bring unused hypothesis into our proofs resulting in having unrelated hypothesis in the proofs. Hence, proofs in a *substructural* logic that has no such structural rules can neither use a hypothesis infinitely many times, nor use an unrelated hypothesis. These logics should be a good candidate to reason about the knowledge of non-idealized agents. As it will be shown, the theorem that deals with the idealization: the problem of logical omniscience, is still provable in such logic. But this substructural version of logical omniscience should not cause us the idealization problems that are faced in classical modal logics. The reason being that our structural rules are under control in such logic, so the proofs in this logic shall not exceed the epistemic capacities of the agents.

In order to study the capabilities of such logics in dealing with epistemic applications, we have chosen to work on a modal linear logic. Linear logic has several more interesting mathematical properties than being a substructural logic. These mathematical properties might also be considered as proper motivations for an epistemic modal logic. In this paper we have only focused on the resource management and substructural properties of linear logic, leaving the study and applications of other properties as a future work . One problem of applying linear logic to epistemic application is the complexity of its proofs. To partly overcome this problem, we have decided to encode it in the Coq proof assistant. The encoding will enable us to state and prove theorems of the modal linear logic using Coq's facilities. To show the capabilities of this encoding in epistemic applications, we will prove two standard theorems of this field using Coq.

3 Modal Linear Logic $KDT4_{lin}$

Linear Logic is a logic introduced by Girard in 1987 [5] as a refinement of classical logic. It is a substructural logic in the sense that it dismisses *contraction* and *weakening* in their original form. A sequent of the form $\Gamma \vdash \Delta$ in linear logic means that resource presented by Γ are to be consumed yielding resources Δ deduced. This makes linear logic a *resource-sensitive* logic. We can also think of the sequent $\Gamma \vdash \Delta$ as a process that consumes the resources Γ to produce the resources Δ. This *resource-sensitive* property of linear logic makes the conjunction and disjunction of classical logic ambiguous. For example We can use $A \wedge B$ both for producing A and also $A \wedge B$ itself(refer to [5] for a more detailed discussion).To overcome these ambiguities, linear logic uses two distinct connectives for each of conjunction and disjunction. They are called multiplicatives. and additives respectively. We write $A \oplus B$ and $A\&B$ for the two connectives for additives, and $A \otimes B$ and $A \parr B$ for the two multiplicatives. Negation is defined by means of the following sequent rules:

$$\text{Negation} \quad \frac{\Gamma \vdash A, \Delta}{\Gamma, A^{\perp} \vdash \Delta}\ Left \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash A^{\perp}, \Delta}\ Right$$

The two multiplicatives are De Morgan duals of each other [2]:

$$\text{Times} \quad \frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \otimes B \vdash \Delta} \otimes L \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma_1 \vdash B, \Delta_1}{\Gamma, \Gamma_1 \vdash A \otimes B, \Delta, \Delta_1} \otimes R$$

$$\text{Par} \quad \frac{\Gamma, A \vdash \Delta \quad \Gamma_1, B \vdash \Delta_1}{\Gamma, \Gamma_1, A \parr B \vdash \Delta, \Delta_1} \parr L \qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \parr B, \Delta} \parr R$$

The same is true for the two additives. Linear implication will be the same as linear deduction and will be denoted by $A \multimap B$. Multiplicatives, additives and linear implication have left and right sequent rules [5]. An infinite resource, i.e. a resource that can be consumed more than once is shown using the linear exponentials and be written as $!A$ and its De Morgan dual $?A$ [2]:

$$\text{Of Course} \quad \frac{\Gamma, A \vdash \Delta}{\Gamma, !A \vdash \Delta} !L \qquad \frac{!\Gamma \vdash B, ?\Delta}{!\Gamma \vdash !A, ?\Delta} !R$$

$$\text{Why Not} \quad \frac{!\Gamma, A \vdash ?\Delta}{!\Gamma, ?A \vdash ?\Delta} ?L \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash ?A, Delta} ?R$$

Exponentials are used to control the structural rules mentioned before. Linear *contraction* and *weakening* are shown below:

$$\textit{Contraction} \quad \frac{\Gamma, !A, !A \vdash \Delta}{\Gamma, !A \vdash \Delta} \; Left \qquad \frac{\Gamma \vdash !A, !A, \Delta}{\Gamma \vdash !A, \Delta} \; Right$$

$$\textit{Weakening} \quad \frac{\Gamma \vdash \Delta}{\Gamma, !A \vdash \Delta} \; Left \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash !A, \Delta} \; Right$$

Modalities can be added to linear logic in many different ways. An example is the view that considers exponentials as a kind of modality enriched with structural rules [1].These different ways result in different combinations of linear logic with modal logic, e.g. some such combinations and their semantics have been studied in [8].As part of our motivation, we are interested in a combination that keeps exponentials and adds modalities on top of them. The resulting logic $KDT4_{lin}$ has an algebraic semantics , which has been proven to be sound and complete[8]. The BNF of this logic is shown below:

$$A ::= a|!A|?A|K_iA|A \otimes A|A \parr A|A \oplus A|A\&A|A \multimap A|A^{\perp}|1|0|\top|\perp$$

where:

- a is an atomic formula

- $1, \bot, \top$, and 0 are the units for $\otimes$, $\parr$, $\&$, and $\oplus$ respectively
- K is the modal operator
- i is a natural number ranging over a denumerably infinite set
- $K_i A$ intuitively means that i knows A.

The sequent rules of this logic are the same as the full propositional classical linear logic in which we take sequents $\Gamma \vdash \Delta$ for lists of formulas. These lists together with the *Exchange* rule, will have the properties of multisets. The modal rules correspond to T, K, D, and S4 axioms of one of the classical Hilbert Style modal logics namely S4:

- Axiom K : $(KA \wedge K(A \rightarrow B)) \rightarrow KB)$
- Axiom D : $\sim K$ *false*
- *Axiom T* : $KA \rightarrow A$
- *Axiom S4* : $KA \rightarrow KKA$

These axioms have been translated into Gentzen sequent rules for classical modal logics [9, 10]. The Gentzen-style modal rules of $KDT4_{lin}$ are the same as that of classical logics. One difference of the sequent version of these rules over the Hilbert axioms is that in the Gentzen modal system, K and D are combined to get the modal rule KD. The left KD rule shown later together with the sequent rules for $\top$ (below):

$$\frac{}{\Gamma \vdash \top}\ \top$$

and the fact that 0 is the dual of $\top$ states the same thing as axiom D, i.e. we don't know any false proposition. The right KD rule is about distribution of knowledge the same way as in the axiom K. In other words, if we can prove B using these hypothesis Γ and A, then if we know the hypothesis, we can prove that we know B. Since not all of the modal axiom are independant of each other, for example D is derivable from T, there is some overlap between their coressponding sequent rules. That is why T and KD right rules are the same. These rules have the same meaning as their Hilbert-style counterparts.

The sequent calculus of $KDT4_lin$ will thus contain the following three modal sequent rules:

$$TRules \quad \frac{\Gamma, A \vdash B, \Delta}{K_i\Gamma, K_iA \vdash B, \Delta}\ Left \qquad \frac{\Gamma, A \vdash B}{K_i\Gamma, K_iA \vdash K_iB}\ Right$$

$$KDRules \quad \frac{\Gamma, A \vdash 0}{K_i\Gamma, K_iA \vdash 0}\ Left \qquad \frac{\Gamma, A \vdash B}{K_i\Gamma, K_iA \vdash K_iB}\ Right$$

$$S4Rules \quad \frac{\Gamma, A \vdash B, \Delta}{\Gamma, K_i A \vdash B, \Delta}\ Left \qquad \frac{K_i\Gamma, K_i A \vdash B}{K_i\Gamma, K_i A \vdash K_i B}\ Right$$

Note that the KD and T rules are instances of a pair of rules with Δ on the right hand side.
In all the modal rules Γ is a multiset of formulas and $K_i\Gamma$ is the multiset $\{K_i A | A \in \Gamma\}$

4 Encoding the Sequent Rules

The logic $KDT4_{lin}$ is encoded in Coq following the two-level meta-reasoning methodology presented in [4].Thus $KDT4_{lin}$ is treated as an *object* logic while the sequent rules are encoded using the Calculus of Inductive Constructs(CIC). We shall first define the connectives of $KDT4_{lin}$ inductively, then augment the grammar. In the next step all the sequent rules of our logic are defined inductively using the previously defined linear proposition type. Multi-sets are implemented using Coq's *list* type.Thus, we shall have some lemmas to work with lists.

4.1 Encoding Linear and Modal Connectives

Following [11], we define inductively a set of linear logic propositions: *MLinProp*, which stands for *Modal LINear PROPosition*. The smallest formulas of our modal linear logic will be the different cases of induction.The definition is shown below:

```
Inductive MLinProp : Set :=
| Implies : (MLinProp) → (MLinProp) → MLinProp
| Times : (MLinProp) → (MLinProp) → MLinProp
| Par : (MLinProp) → (MLinProp) → MLinProp
| Plus : (MLinProp) → (MLinProp) → MLinProp
| With : (MLinProp) → (MLinProp) → MLinProp
| OfCourse : (MLinProp) → MLinProp
| WhyNot : (MLinProp) → MLinProp
| Box : (nat) → (list MLinProp)→(list MLinProp)
| Negation : (MLinProp) → MLinProp
| One : MLinProp
| Zero : MLinProp
|⊥ : MLinProp
| ⊤ : MLinProp
```

Now we can use our *MLinProp* as a Coq type. We can define variables of this type. For example we can define A and B as modal linear propositions, and D as a list of Modal linear proposition:

```
Variable A, B : MLinProp.  Variable D : (list MLinProp).
```

We can also define predicates over this type. For example red is a 1-ary modal linear predicate:

```
Variable red:  nat → MLinProp.
```

Note that all of the connectives input linear propositions, but the modality Box, which takes as input a list of linear propositions. This is because the modality sequent rules mentioned before. These rules need our modality to operates over a list of formulas rather than a single formula. The modality is also an indexed modality, making our logic a multi-modal linear logic. A multi-modal logic is a logic an indexed modality One of the applications of such a modality is the epistemic application as will be discussed later. In this application we want to be able to reason about the knowledge of a group of agents. This means that each modality K_i, has to be indexed to express the knowledge of each agent. For example $K_1 D$ intuitively means that *agent one knows that* D i.e. he knows all of the formulas of the list D. The modality operator can be seen as a binary operator with two operands: an integer! and a list of formulas.

Using Coq's syntax definition and pretty-printing facilities, we can give a notation to each of our modal linear connectives. This will allows us to infix and prefix our connectives.The Coq code for Bang, Times, and Box is given below. We are augmenting the grammar rules and give pretty-printing rules to represent Bang as "!", Times as "*", and Box as "K".The reader is assumed to be familiar with the syntax of these Coq commands (see section 6.7.3 and 6.7.4 of Coq Manual).

```
Grammar command command2 :=
OfCourse [``!'' command2($c)] →[⟨⟨(OfCourse $c)⟩⟩].
Syntax constr level 2:
[⟨⟨(OfCourse $c)⟩⟩]→[``!'' $c].
Grammar command command6 :=
Times [command5($c1) ``*'' command6($c2)] → [⟨⟨(Times $c1 $c2)⟩⟩].
Box [command5($c1) ``K'' command6($c2)] → [⟨⟨(Box $c1 $c2)⟩⟩].
Syntax constr level 6:
PTimes [⟨⟨(Times $c1 $c2)⟩⟩] → [ $c1:L "*" $c2:E ].
Syntax constr level 6:
PBox [⟨⟨(Box $c1 $c2)⟩⟩]→ [ $c1:L "K" $c2:E ].
```

The notation for all of our modal linear connectives is given in the table below for further reference. The full Coq code has been given in the appendix.

Connective	Symbol	Syntax in Coq	Example
Times	$\otimes$	**	$A**B$
Par	$\parr$	%	$A\%\%B$
Plus	$\oplus$	$\oplus$	$A++B$
With	&	&	$A\&B$
Box	K	K	${}_iKD$
OfCourse	!	!	$!A$
Implies	$\multimap$	$\multimap$	$A \multimap B$

4.2 Encoding Linear and Modal Sequent Rules

In the second phase of our encoding, we will implement the sequent calculus of our modal linear logic. The sequent rules are defined inductively. The induction is made on the linear sequent relation $\Gamma \vdash \Delta$. The sequent relation *LinCons* has been represented as a 2-ary function. It takes two arguments as input: the hypothesisΓ and the conclusion Δ. Remember that Γ and Δ are implemented as lists of formulas. These lists together with the exchange and permutation rules will act as multisets. The output of the linear sequent relation *LinCons* is either true or false and is defined as a Coq proposition *Prop*. The Coq code for *LinCons* is shown below:

```
Inductive LinCons :  (list MLinProp) → (list MLinProp) →
Prop :=
```

The connective "$\vdash$" is defined as a binary operator with a low precedence using the Coq Syntax and pretty-printing commands:

```
Grammar command command9 :=
LinCons [command8($t1) ``⊢'' command9($t2)] → [⟨⟨(LinCons $t1 $t2)⟩⟩].
Syntax constr level 9:
PLinCons [⟨⟨(LinCons $t1 $t2)⟩⟩]→ [ $t1 ``⊢'' $t2 ].
```

The axiom and the sequent rules of the modal linear logic will be the cases of the induction. They are added individually. For example the axiom *Identity* is added as follows:

```
Identity :
(A :  MLinProp)
(`A ⊢ `A)
```

The sequents of our system are of the form $D1 \hat{} `A \vdash D2 \hat{} `B$, where $D1$ and $D2$ are lists of formulas of type *MLinProp*, and A and B are formulas of the type *MLinProp*. Note that we have lists on both sides of the sequent. In other words our logic is not intuitionistic as opposed to that of [11]. This helps us to encode all the connectives of linear logic including Par '$\parr$' Following the encoding of [11], two symbols $\hat{}$ and ` are used to work with lists in Coq; $\hat{}$

is used to concat two lists and ‘ presents a singleton list. For example, $D1 \,\hat{}\, `A$ concatenates two list $D1$ and the singleton A. The empty list will be shown as*Empty*. Logical and structural rules of modal linear logic are added next. These rules are coded using Coq's implication $\rightarrow$ for deduction. For example the *Cut* rule [2]:

$$\frac{\Gamma_1 \vdash A, \Delta_1 \quad \Gamma_2, A \vdash \Delta_2}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ Cut$$

is coded as below:

```
| Cut :
(A, B : MLinProp)(D1, D2, D3, D4 : (list MLinProp))
((D1 ⊢ D3 ^ `A) → (D2 ^ `A ⊢ D4) → (D1 ^ D2 ⊢ D3, D4))
```

As examples of logical rules, the Coq code for Par Left and Times Right is shown below:

```
| ParLeft :
(A, B, C1 , C2 : MLinProp)(D1, D2 , D3,D4 : (list MLinProp))
((D1 ^ ‘A ⊢ ‘ C1 ^ D3) → (D2 ^ ‘B ⊢ ‘C2 ^ D4) → (D1 ^ D2 ^
‘(A%%B) ⊢ ‘C1 ^ ‘C2 ^ D3 ^ D4))
|TimesRight :
(A, B : MLinProp)(D1, D2 , D3 , D4: (list MLinProp))
((D1 ⊢ ‘A ^ D3) → (D2 ⊢ ‘B ^ D4) → (D1 ^ D2 ⊢ ‘(A
** B) ^ D3 ^ D4))
```

The modal sequent rules are KD, T, and S4. The different thing about these rules is that the modal operator has two operands: an index i and a list of formulas D. K_iD will be shown as iKD in Coq.For example the KD rule below:

$$\frac{\Gamma, A \vdash B}{iK\Gamma, iKA \vdash iKB}\ KD$$

will be code as:

```
| KDRule :
(i : nat)(A, B : MLinProp)(D : (list MLinProp))
((D ^ ‘A ⊢ ‘B) → (‘(iKD) ^ ‘(iK‘A) ⊢ ‘(iK‘B)))
```

4.3 Some Lemmas to Work with Lists

For the reason of clarity, we have chosen to work with sequents with distinguished formulas to the left and right hand sides of "⊢":

$$\Gamma, A \vdash B, \Delta$$

But most of the time Γ and Δ are empty lists and the sequent is of the form:

$$A \vdash B$$

Sequents of this form cause us problem because encoded rules of our logic cannot be applied to them. As an example consider the following deduction which is a valid one:

$$\frac{A \vdash A}{A \vdash A \oplus B}\ \oplus R1$$

We will face difficulties in proving this deduction because it does not match the general format of the encoded seqeunts. Thus no rule can be applied to $A \vdash A \oplus B$ whereas $\oplus R$ should be applicable. To solve the problem we will have to add Nil lists to the left hand side of the leading formulas A and $A \oplus B$. Thus the above deduction will look like the following after these changes:

$$\frac{\mathrm{E}mpty, A \vdash \mathrm{E}mpty, A}{\mathrm{E}mpty, A \vdash \mathrm{E}mpty, A \oplus B}\ \oplus R1$$

This will be done using two lemmas: *AddNilLeft* and *AddNilRight*. *AddNilFront* is shown below:

$\mathtt{L}emma\mathtt{A}ddNilLeft : (D1, D2 : (\mathtt{l}ist\ \ MLinProp))((\mathtt{E}mpty\ \hat{}\ D1 \vdash D2) \rightarrow (D1 \vdash D2)).$

Each of these lemmas has a dual to eliminate the added Nils if necessary. Eliminating Nils will be done using *ElimNilLeft* and *ElimNilRight* lemmas. *ElimNilRight* is shown below.

$\texttt{L}emma\texttt{E}limNilRight : (D1, D2 : (\texttt{list}\texttt{M}LinProp))((D1 \vdash D2) \rightarrow (D1 \vdash \texttt{E}mpty \ \hat{} \ D2)).$

There are several other approaches to this problem. For example we could encode our sequents in such a way that left and right hand sides of the "$\vdash$" consist of only one list and no distinguished formula. Then the problem would be solved by making a singleton list out of the single formulas that appear on the RHS and LHS of "$\vdash$". While in this approach, list concatenation and singleton lists could be dealt with the same way as the encoding of [11].

5 Epistemic Applications

5.1 Proving a monomodal theorem

In this section we are going to prove a common property of most of the modal logics, i.e. *closure under material implication*:

$$K_1(A \multimap B) \vdash K_1 A \multimap K_1 B$$

To make the theorem more interesting, we slightly changed it to the following form:

$$K_1 A, K_1(A \multimap B) \vdash K_1 B$$

In informal terms, if an agent knows A and $A \multimap B$ then he also knows B. The proof tree is given below:

$$\dfrac{\dfrac{A \vdash A \quad B \vdash B}{A, (A \multimap B) \vdash B} \multimap L}{K_1 A, K_1 A \multimap B \vdash K_1 B} \text{KDRule}$$

The proof in Coq is done in the same steps as the proof tree above. But we had to add *Nil* to the left of our sequents to be able to apply the *Implies Left* rule and the *Identity* axiom.The coq code is shown below:

```
Intros.
Apply KDRule.
Apply AddNilLeft.
Apply ImpliesLeft.
Apply AddNilLeft.
Apply Identity.
Apply Identity.
```

Note that the above theorem says that agents know all the logical consequences of their knowledge . This makes the agents of a classical modal logic idealized. But our modal logic by being substructural only allows agents to do

proofs that do not exceed their cognitive capacities. It has embedded exponentials in structural rules to control the proofs so that they will not become unfeasible. This is one of the measures that make our logic a better candidate for an epistemic logic.

5.2 Proving a multimodal theorem

A standard puzzle of multi-modal logics is the The king, three wise men and 5 hats puzzle. It goes like this: a king has got three wise men and 5 hats: 2 green and 3 red. He asks the wise men to close their eyes and puts a hat on the head of each of them. Then asks them to open their eyes and poses a question to each of them in order. He asks the first man: 'Do you know the color of your hat?' He answers: 'No'. The same question is asked from the second man and he, too, answers No. But when the third man is asked the same question, he answers: ='Yes! The color of my hat is red'. How this is possible? We will show here that this conclusion can be made based on the information provided by the answers of previous wise men and using the sequent rules of our modal linear logic.In more formal terms we have: if agent three knows that agent one does not know the color of his hat, and he knows that agent two does not know the color of his hat and moreover he knows that agent two knows that agent one does not know the color of his hat, he will conclude the color of his own hat. Agent 3 ,therefore, knows the color of the hats of the other agents, the following three items that help him together with a good number of assumptions and some lemmas to conclude the color of his own hat, which is red:

1. Agent one does not know the color of his hat.
2. Agent two does not know the color of his hat.
3. Agent two knows that agent one does not know the color of his hat.

These information will help agent three to conclude that the color of his own hat is red. From (1) it can be concluded that at least one of the agents two and three wear a red hat. Because if both of them had green hats and since we only have two green hats, agent one would know the color of his hat. So a corollary of (1) is that agents 2 and three both know the following fact: At least one of agents two or three wears a red hat (or both of them do) This fact, together with (2) and (3) above help agent three to conclude that his hat is red. The fact that agent two does not know the color of his hat shows that agent three is not wearing a green hat. Because if this was the case, agent two, who knows that at least one of them is wearing a red hat, would have easily concluded the color of his own hat. In order to prove this theorem in the Coq, we have to define three agents, two color predicates and one definition.

1. Three agents:

```
agent1, agent2, agent3 :  nat.
```

2. Two color predicates:

- `(red i): the color of the hat of ith agent is red`
- `(green i): the color of the hat of ith agent is green`

3. Definition

   When each agent knows the color of his hat, it means he knows whether it is red or green. This can be shown using the additive Plus because it expresses a choice between two cases, where both of the cases cannot happen at the same time.

   (Lhat i): agent i knows that his hat is either red or green.

   or in Coq terms:

   `Definition Lhat := [i: nat](K_i '(red i)) ⊕ (K_i '(green i)).`

We will use the same proof method as of Lescanne[7] with 6 axioms but in a linear logic environment:

1. AOne:Each hat is either red or green. This can again be shown using the additive Plus because (green i) and (red i) cannot both happen at the same time, i.e. each hat cannot be both red and green at the same time.

   `(i:nat)(Empty ⊢ '((green i)⊕ (red i))).`

2. ATwo:ATwo says that if two agents wear a green hat then the third one wears a red one. In this axiom, as opposed to the previous one, we want to be able to express that two cases happen at the same time, i.e. both agents wear a green hat. One of the Multiplicative connector seems a good option. We are going to use Par.

   `Axiom ATwo : ('((green agent2) ⅋ (green agent3)) ⊢ '(red agent1)).`

3. AThree: If agent2 has a green hat, then agent one knows it. The reason is obvious because he is seeing the hat of agent2.

   `('(green agent2) ⊢ '(agent1 K '(green agent2))).`

4. AFour: If agent3 has a green hat, then agent one knows it. The reason is obvious because he is seeing the hat of agent3.

   `('(green agent3) ⊢ '(agent1 K '(green agent3))).`

5. AFive : If an agent is wearing a red hat, then he is not wearing a green one.

   `('(red i) ⊢ '(Not (green i))).`

6. ASix : If an agent is wearing a green hat, then he is not wearing a red one.

   `('(green i)) ⊢ '(Not (red i))).`

The theroem to be proved in sequent calculus is:

(agent2 K (Not (Lhat agent1))), (Not (Lhat agent2)) $\vdash$ (red agent3)

Or in Coq terms:

```
Theorem ThirdKnows :
(‘(Not (Lhat agent2)) ^ (‘(agent2 K ‘(Not(Lhat agent1)))) ⊢ ‘(red
agent3)).
```

The proof is done mostly with cuts. The proof tree and the Coq code is given in the appendix.

6 Discussion and Conclusion

The first version of this encoding was done in an intuitionistic fragment of modal linear logic. In that fragment sequents were limited to have only single formulas on their right hand side. Thus the first encoding of our logic had lists only in the left hand side of sequents and the right hand side contained only a single formula. This fragment did not have all the connectors of linear logic. In particular it missed Par $\parr$, the dual of Times $\otimes$. The reason being that sequent rules for Par, shown below are not intuitionistic:

$$Par \quad \frac{\Gamma \vdash A, B}{\Gamma \vdash A \parr B}\ R \qquad \frac{\Gamma_1, A \vdash C \quad \Gamma_2, B \vdash D}{\Gamma_1, \Gamma_2, A \parr B \vdash C, D}\ L$$

The problem with the intuitionistic fragment that dismissed Par was that in the proof of the puzzle at some stage we had to work with the dual of Times. Our choice of linear connectives in the encoding of the puzzle shown in the previous section, led us to prove the following sequent in the proccess of the proof of puzzle:

$$\texttt{Not ((red 1)} \otimes \texttt{(red 2))} \vdash \texttt{(green 1)} \parr \texttt{(green 2)}$$

The proof seemed impossible in the fragment without $\parr$. In order to be able to solve the puzzle and to keep to our presented encoding, we decided to work with the full fragment of modal linear logic. Nevertheless, one way to stay in an intuitionistic fragment would be to re-phrase the puzzle and the axioms using the intuitionistic version pf $\parr$ introduced in [2]. Changing to a classical fragment, we had to add lists to both sides of our sequents:

$$D1 \ ^\frown \ ‘A \vdash D2 \ ^\frown \ ‘B$$

As presented before, the puzzle was proven in this fragment. Witnessing this experiment, we believe that classical modal linear logic is a better candidate for such epistemic applications .

As conclusion, we have encoded a classical modal linear logic KDT_{lin} in higher order logic using proof assistant *Coq*. The encoding was done following the tow-level meta-reasoning in *Coq* presented in [4] and used before in a

previous encoding of intuitionistic linear logic in *Coq* [11]. The logic has been previously developed [8] by adding KD, T, and S4 modalities other than exponentials to Girard's classical linear logic. The encoding provided us with *Coq's* facilities to show how this logic can be successfully used in epistemic applications. However, the main result of this work was to show the feasibility of *Coq* as a proof assistant for the classical modal linear logic. The epistemic examples presented here demonstrated the benefits's of the encoding specially in dealing with lists of formulas on both sides of the sequent relation. A good suggestion for a further project would be to try to prove other puzzles of epistemic logic such as 'Muddy Children' or the puzzles that cannot be solved in classical epistemic logics and are only solvable in epistemic linear logic. Proving the latter puzzles will show the differences of linear logic over classical logics and thus will provide us with benifits of using linear logic as an epistemic logic over classical ones. One could also use *Coq* to compare the linear logic proofs with their classical logic counterparts. There are *Coq* encoding's for classical epistemic logics both in sequent and Hilbert style systems [7]. Complexity analysis and proof automation using *Coq's* mechanisms are yet other further project that can be built on this work.

References

[1] A. Avron, 'Syntax and Semantics of Linear Logic', *Theoretical Computer Science* 57, pp.161-184, 1988.

[2] T.Brauner , and V. de Paiva , 'Cut-Elimination for Full Intuitionistic Linear Logic', `www.bricks.dk/RS/96/10`

[3] J. Dubucs, and M. Marion, 'Radical Antirealism and Substructural Logics', to appear in *Proceedings of LMPS99*, Dordrecht, Kluwer, 2002.

[4] A. Felty, 'Two-Level Meta0Reasoning in Coq', *Fifteenth International Conference on Theorem Proving in Higher Order Logics*, Springer-Verlag LNCS 2410, 2002

[5] J-Y. Girard, 'Linear Logic', *Theoretical Computer Science*, 50, pp.1-102, 1987

[6] J. Hintikka, *Knowledge and Belief, An Introduction to the Logic of Two Notions*, N.Y., Cornell University Press, 1969

[7] P. Lescanne. ' Epistemic Logic in Higher Order Logic', `http://www.ens-lyon.fr/LIP/Pub/rr2001.html`, 2001.

[8] A. Martin. *Modal and Fixpoint Linear Logic*, Masters Thesis, Department ofo Mathematics and Statistics, University of Ottawa, 2002.

[9] M. Ohnishi, and K. Matsumoto, 'Gentzen Method in Modal Calculi, I, *Osaka Mathematical Journal 9*, pp. 113-130, 1957

[10] M. Ohnishi, and K. Matsumoto, 'Gentzen Method in Modal Calculi, II, *Osaka Mathematical Journal 11*, pp. 115-120, 1959

[11] J. Powers, and C. Webster, 'Working with Linear Logic in Coq',12th International Conference on Theorem Proving in Higher Order Logics, 1999.

7 Appendix

7.1 Proof Tree

$$\dfrac{\dfrac{\Pi_1}{K_2(Not(Lhat1)) \vdash (red2) \otimes (red3)} \quad \dfrac{\dfrac{}{Not(Lhat2), (red2) \otimes (red3) \vdash (red2) \otimes (red3)}\ Identity}{Not(Lhat2), (red2) \otimes (red3) \vdash (red3)}\ \text{CUT} \quad \dfrac{\dfrac{}{(red2) \otimes (red3) \vdash (red3)}\ Identity}{(red2) \otimes (red3) \vdash (red3)}\ \otimes\text{L}}{K_2(Not(Lhat1)), Not(Lhat2) \vdash (red3)}\ \text{CUT}$$

Π_1 :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(red1) \vdash (red1)}\ Identity}{(red1) \vdash (red1) \oplus (green1)}\ \oplus\text{R}}{K_1(red1) \vdash K_1((red1) \oplus (green1))}\ \text{KD}}{K_1(red1) \vdash (Lhat1)}\ \text{Unfold}}{Not(Lhat1) \vdash Not(K_1(red1))}\ \text{Negation} \quad \dfrac{\Pi_2}{Not(K_1(red1) \vdash (red2) \otimes (red3))}}{Not(Lhat1) \vdash (red2) \otimes (red3)}\ \text{CUT}}{K_2(Not(Lhat1)) \vdash (red2) \otimes (red3)}\ \text{S4}$$

Π_2 :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(green2) \vdash K_1(green2)}\ \text{A3} \quad \dfrac{}{(green3) \vdash K_1(green3)}\ \text{A4}}{(green2) \parr (green3) \vdash K_1(green2), K_1(green3)}\ \parr\text{L}}{(green2) \parr (green3) \vdash K_1(green2) \parr K_1(green3)}\ \parr\text{R} \quad \dfrac{\dfrac{}{(green2) \parr (green3) \vdash (red1)}\ \text{A2}}{K_1(green2) \parr K_1(green3) \vdash K_1(red1)}\ \text{T}}{(green2) \parr (green3) \vdash K_1(red1)}\ \text{CUT}}{Not(K_1(red1)) \vdash Not((green2) \parr (green3))}\ \text{Negation} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{(red2) \vdash Not(green2)}\ \text{A5}}{(red2), (green2) \vdash Empty}\ \text{Negation} \quad \dfrac{\dfrac{}{(red3) \vdash Not(green3)}\ \text{A5}}{(red3), (green3) \vdash Empty}\ \text{Negati}}{(red2), (red3), (green2), (green3) \vdash Empty}\ \text{CUT}}{(red2), (red3) \vdash Not((green2) \parr (green3))}\ \text{Negation}}{(red2) \otimes (red3) \vdash Not((green2) \parr (green3))}\ \otimes\text{L}}{Not(K_1(red1)) \vdash (red2) \otimes (red3)}\ \text{CUT}$$

7.2 Coq Code

```
Section Hats.
Load MALL.
Variables red, green :  nat → MLinProp.
Variables agent1, agent2, agent3 :  nat.
Definition Lhat := [i:nat](i K `(red i)) ++ (i K `(green i)).
Axiom AOne :
(i:nat)(D : (list MLinProp))
( D ⊢ ` ((green i) %% (red i))).
Axiom ATwo :
(`((green agent2) %% (green agent3)) ⊢ `(agent1 K `(red agent1))).
Axiom AThree :
(`(green agent2) ⊢ `(agent1 K `(green agent2))).
Axiom AFour :
(`(green agent3) ⊢ `(agent1 K `(green agent3))).
Axiom AFive :
(i :  nat)(`(red i) ⊢ `(Not (green i))).
Axiom ASix :
(i :  nat)(`(green i) ⊢ `(Not (red i))).
Lemma Duals :
(i , j :  nat)
(`(Not ((green i)Proof.
Apply TimesLeft.
Apply AddNilRight.
Apply NegationRight.
Apply NegationRight.
Apply ParLeft.
Apply NegationRight.
Apply AFive.
Apply NegationRight.
Apply AFive.
Qed.
(* Main Theorem *)
Theorem ThirdKnows :
(`(Not (Lhat agent2))  ^  (`(agent2 K `(Not(Lhat agent1))))⊢ `(red agent3)).
(* Proof *)
Intros.
Apply Cut with (Times (red agent2) (red agent3)).
Apply AddNilLeft.
Apply S4Rule1.
Apply Cut with (Negation (agent1 K `(red agent1))).
Apply AddNilLeft.
Apply NegationLeft.
Apply AddNilRight.
Apply ExchangeRight.
```

```
Apply ElimNilRight.
Apply NegationRight.
Unfold Lhat.
Apply PlusRight1.
Apply Identity.
Apply Cut with (Negation (Par (green agent2) (green agent3))).
Apply AddNilLeft.
Apply NegationLeft.
Apply AddNilRight.
Apply ExchangeRight.
Apply ElimNilRight.
Apply NegationRight.
Apply ElimNilLeft.
Apply ATwo.
Apply ElimNilLeft.
Apply Duals.
Apply Cut with (red agent3).
Apply AddNilLeft.
Apply TimesLeft.
Apply ElimNilLeft.
Apply Identity.
Apply Identity.
End Hats.
```

Representing RSL Specifications in Isabelle/HOL

Morten P. Lindegaard

Informatics and Mathematical Modelling, Technical University of Denmark,
DK-2800 Kgs. Lyngby, Denmark
mpl@imm.dtu.dk

Abstract. The RAISE development method is a software development method with an associated specification language, RSL. An approach to increasing proof support for the RAISE method is to translate RSL specifications and proof obligations into higher-order logic (HOL) and use a theorem prover which supports HOL. The higher-order logic instance Isabelle/HOL of the generic theorem prover Isabelle is used for this purpose. A translation based on a denotational semantics for a subset of RSL is presented, and experiences with translating a larger subset of RSL are discussed. RSL proof rules are also translated to Isabelle/HOL and proved as theorems.

1 Introduction

The RAISE Specification Language (RSL) [11] is a wide-spectrum specification language associated with a development method [12] based on step-wise refinement. Proving properties of specifications is important to the method and software tools support this task. However, increased mechanization of the proofs is desired, and in this paper we shall report on a new development of proof support for RAISE.

An approach to increasing mechanization of proofs needed in the RAISE method is to use the theorem prover Isabelle [10] which offers support for higher-order logic (HOL). In order to use this theorem prover, specifications and proof-obligations written in RSL are translated to Isabelle/HOL and the proofs are carried out using Isabelle.

RSL has many features such as programming variables, non-determinism, and channels which complicate translation to HOL. However, it is our experience that a simpler subset of RSL suffices for a wide variety of formal developments and therefore will be of interest. Consequently, an applicative, deterministic subset of RSL is considered at first.

The translation must be done in a well-founded way to ensure that properties proved for the translated specifications in HOL also hold for the original specifications in RSL. We base the translation/representation on the denotational semantics of RSL [7], i.e., we have defined a denotational semantics for a suitable subset of RSL such that it conforms with the original semantics for the full language.

The concept of institutions [13] may be used as a framework for translating between logics. Institutions formalize the informal notion of a logical system with signatures, sentences, models, and a satisfaction relation. An institution representation is a morphism between two institutions. It consists of morphisms between the components of the institutions, e.g., a morphism which translates sentences. When certain requirements are met, a theorem prover may safely be reused when translating specifications along the institution representation.

The denotational semantics is used to define an institution for the subset of RSL. This institutional description enables the definition of an institution representation (including a translation of sentences) that makes reuse of Isabelle/HOL possible.

Details of the definition of the institution and institution representation are beyond the scope of this paper and are given in a paper [6] currently under construction. In this paper, focus will be on how to represent various features of RSL in Isabelle/HOL.

1.1 Outline of the Paper

Section 2 introduces RSL, Isabelle/HOL, different approaches to reusing Isabelle, and the use of institutions. In Sect. 3, the translation from a subset of RSL to Isabelle/HOL is described, and an example is presented in Sect. 4. Possibilities for translating a larger subset of RSL are discussed in Sect. 5, before achievements and future work are summarized in Sect. 6 which concludes the paper.

2 Background

2.1 The RAISE Specification Language (RSL)

In this section, we shall briefly touch upon RSL specifications, semantics, tool support, and the RAISE Development Method.

RSL Specifications A basic RSL specification consists of declarations of types, values, axioms, variables, and channels.

RSL has a series of built-in types and type constructors. User-declared types may be introduced as abstract sort types or model-oriented types built from the basic types. Values denote constants and functions that are specified by axioms. Functions may access variables and communicate on channels, i.e., imperative aspects and process behaviour may be described. Moreover, language constructs known from programming languages as well as logical connectives, quantifiers and non-determinism are part of RSL.

The applicative, deterministic subset of RSL considered for translation comprises declarations of types, values, and axioms. In Sect. 3, the subset is described in detail together with the translation.

Specifications may be built from other specifications by renaming declared entities, hiding declared entities, or adding more declarations. Moreover, specifications may be parameterized.

Semantics The denotational semantics of RSL is defined in [7].

RSL facilitates description of both datatypes and process behaviour. Other languages with similar features usually combine two different formalisms such that a specification must contain a part specifying data and a part specifying process behaviour. RSL, on the other hand, has fully integrated datatypes and process behaviour both in the syntax and the semantics. Consequently, a value expression is viewed as denoting a function from a store (containing the state of the variables) to a process! As an example, the value expression **true** denotes the function that takes a store to the process that immediately terminates without side-effects (without changing the store) and returns the boolean value $True$. A process may not terminate, in which case it is said to be divergent.

The semantics of an RSL specification is the collection of the models that satisfy the axioms of the specification, i.e., all the models that interpret the declared entities as prescribed by the axioms.

Proof System and Tools RSL also has a proof system [12] which has been implemented in the justification editor (part of the RAISE tools [3]). Proofs are developed manually. However, the editor shows applicable rules and does some simplification.

A new tool suite including a translator from RSL to PVS has been developed at UNI/IIST [4]. The translator translates an applicative, deterministic subset comprising total functions and declarations of free datatypes. RSL constructs are translated to similar PVS constructs and it is informally argued that the translation is sound. However, the denotational semantics is not considered, nor is a formal framework for relating logics.

The RAISE Development Method RSL is used in connection with the RAISE development method in which step-wise refinement is important. In a series of development steps, an initial abstract specification is refined into a concrete specification suited for implementation. There is a refinement relation between specifications. A specification SP_r refines a specification SP if (*i*) SP_r defines at least the same types and values as SP, and (*ii*) the axioms of SP are consequences of SP_r such that properties are preserved. Consequently, a specification which is the result of a series of refinements has the properties of the initial specification. Refinement may be formally proved (or informally *justified*).

Initial specifications are often applicative since reasoning about applicative specifications is easier than reasoning about imperative specifications. Developing an imperative specification from an applicative specification is not a formal refinement. However, it can be done in a systematic way which is not likely to introduce errors.

2.2 Isabelle/HOL

Isabelle [10] is a generic theorem prover which has been instantiated to offer proof support for a number of logics, e.g., higher-order logic [9]. Instantiating Isabelle amounts to encoding the object-logic in Isabelle's meta-logic.

2.3 Approaches to Using Isabelle

An approach to using Isabelle to provide proof support for RSL is to instantiate Isabelle to RSL by encoding the RSL proof rules within the meta-logic of Isabelle. Another approach is to represent RSL specifications in Isabelle/HOL. These approaches are discussed below.

Instantiating Isabelle The approach of instantiating Isabelle to RSL by encoding the RSL proof rules within the meta-logic of Isabelle would indeed provide proof support for RSL, but it would not bring the denotational semantics into play and we would not be able to use institutions nor use our work to relate RSL proof rules with the denotational semantics. Furthermore, proof support for the basic datatypes, such as integers, should be built from scratch.

Representing RSL in Isabelle/HOL Instead of instantiating Isabelle to RSL, we represent RSL specifications in Isabelle/HOL. This could be done as a deep embedding in which the RSL datatypes and abstract syntax is developed in Isabelle/HOL. However, it is preferable to be able to reuse the datatypes (e.g. the integers) of Isabelle/HOL and the proof support already available for them. Although we reuse datatypes, our representation/translation is based on the semantics of RSL, and consequently, it may be seen as a shallow embedding encoding the semantics of RSL.

When translating RSL specifications to Isabelle/HOL, we can either represent RSL language constructs by similar Isabelle/HOL constructs, or we can let the representation in Isabelle/HOL correspond closely to the denotational semantics of RSL. The advantage of making the translation correspond closely to the denotational semantics is that it makes it easier to justify that it really is part of an institution representation (see the next section) with the desired properties. The disadvantage is that the result of translating an RSL value expression is a HOL term which may be very difficult to understand. However, it is possible to translate to simpler HOL terms that are equivalent to the complicated HOL terms expressing the RSL semantics directly.

2.4 Institutions as a Framework

In this section, we give a brief, informal out-line of the notion of institutions, mappings between institutions, specifications, and a result about reusing theorem provers. This provides a theoretical framework for translating RSL specifications to Isabelle/HOL such that Isabelle can be used.

Institutions The concept of an *institution* formalizes the informal notion of a logical system. An institution consists of signatures, sentences, models, and a satisfaction relation between models and sentences.

An institution [6] for an applicative, deterministic subset of RSL has been defined. The signatures contain user-declared types and values; sentences are formed using the traditional grammar of the language; and models are structures that interpret signatures. The satisfaction relation between models and sentences is defined by the denotational semantics of RSL.

An institution [1] for HOL has also been given on the basis of the model-theoretic semantics given in [5].

Institution Representations Having defined institutions for RSL and HOL, we can define mappings between them. The concept of an *institution representation* describes a mapping between two institutions.

An institution representation from RSL to HOL consists of three mappings. RSL signatures are mapped to HOL theories (signatures and definitions/axioms), RSL sentences are mapped to HOL sentences (terms), and HOL models are mapped to RSL models. The mappings must relate to each other in a certain way such that satisfaction is preserved when mapping models and sentences accordingly.

RSL signatures are mapped to theories in order to ensure a certain interpretation of entities in a HOL signature and thereby restrict attention to certain models.

Specifications and Reusing Theorem Provers Specifications can be defined in terms of signatures and sentences of an institution. A basic (flat) specification consists of a signature and a set of sentences (this corresponds to declarations of types, values, and axioms).

The mappings of signatures and sentences defined in an institution representation may be used to translate specifications of one institution to specifications of another institution (this corresponds to translating declarations of types, values, and axioms).

If an institution representation satisfies certain conditions, it can be used to translate basic specifications such that a theorem prover can be reused [8]. In our case, the mapping of HOL models to RSL models must be surjective.

Since Isabelle/HOL is based on the encoding of higher-order logic given in the HOL-system [5], we assume that the proof system implemented by Isabelle/HOL is sound with respect to the model-oriented semantics given in [5]. This, together with the institution representation, makes safe reuse of Isabelle/HOL possible.

3 Representing RSL in Isabelle/HOL

In this section, we present the core of the translation as well as the definitions in Isabelle/HOL facilitating the translation. In the presentation, Isabelle/HOL details about infix declarations and operator priorities are left out.

3.1 Semantic Domains and Operators

The translation of RSL sentences follows the denotational semantics of RSL. The essence of the translation is that an RSL value expression is translated to a HOL term which expresses the semantics of the value expression.

The key to understanding the semantics of applicative, deterministic RSL is that a value expression represents a "process" that (*i*) terminates immediately and deterministically with a result, or (*ii*) does not terminate. In Isabelle/HOL, a datatype is used to express this notion of simple processes:

```
datatype 'a Process = Proc 'a | Chaos
```

Semantic operators from the denotational semantics are also defined:

```
consts pipe :: "'a Process => ('a => 'b Process) => 'b Process"

primrec
  pipe_proc : "Proc x pipe f = f x"
  pipe_chaos : "Chaos pipe f  = Chaos"

constdefs
  v2p :: "'a => 'a Process"
  "v2p x == Proc x"
```

The operator *pipe* takes a process and a function as arguments, and is used to handle divergence. If the given process is divergent, the result is divergent. If the given process terminates with a value, the result is found by applying the function to that value.

Given a value, the operator *v2p* yields the process that terminates immediately with the value as the result.

These semantic operators stem from the original semantics for full RSL, but they have been simplified to accomodate exactly the features we have chosen to consider.

3.2 Translating Specifications

In RSL, specifications are built from *class expressions* which contain declarations of types, values, axioms, etc. A specification is essentially a named class expression also known as a *scheme*. We only translate basic (i.e. not structured) specifications of the form:

scheme A =
  class
    ...
  end

The theory `RSL` contains all the definitions needed to facilitate the translation. It is used as a stub such that the result of translating an RSL scheme is an Isabelle/HOL theory of the form:

```
theory A = RSL :
  ...
end
```

In RSL, the order of the declarations in a specification does not matter. However, this is not the case in Isabelle/HOL where entities must be defined before used. Consequently, declarations are translated in the following order: (*i*) type declarations, (*ii*) value declarations, (*iii*) axiom declarations.

3.3 Translating Type Declarations and Type Expressions

Type Declarations The considered subset comprises two kinds of type declarations: abbreviation types and sort types.

An *abbreviation type* declares a new name for another type. Whereas the declaration of a *sort type* declares the name of the type without specifying it any further. Examples are:

type T = type_expr
type S

where T is introduced as another name for the type denoted by the type expression type_expr, and S is a sort.

Type expressions are formed from the basic types, user-declared types, and the type operator $\xrightarrow{\sim}$. In the example, S could occur in type_expr. However, declarations of abbreviation types are not allowed to be cyclic.

Since the order of declarations is important in Isabelle but not in RSL, the type declarations of a specification must be translated with care. If S occurs in type_expr, the sort type declaration must be translated before the abbreviation type declaration. Consequently, declarations of sort types are always translated first. The abbreviation type declarations are then translated in a loop: if the right-hand side only contains names of types that are already translated (or are built-in), then the declaration is translated. Since abbreviation type declarations are assumed not to be cyclic, this process of translation terminates.

The example type declarations above are translated to:

```
typedecl S
types T = type_expr
```

where `type_expr` is the result of translating the RSL type expression as described below.

Type Expressions The basic RSL types **Unit**, **Bool**, **Real**, and **Int** are represented by the types `unit`, `bool`, `real`, and `int`.

The RSL function types of the form type_expr_1 $\xrightarrow{\sim}$ type_expr_2 are represented by `type_expr_1 => (type_expr_2 Process)`. This way of representing the partial functions of RSL by the total functions of HOL and the notion of processes corresponds exactly to the way functions are viewed in the denotational semantics of RSL.

Names of user-declared types are translated to the same names.

3.4 Translating Value Declarations

A value is declared with a name and a type. A value declaration of the form

value id : type_expr

is translated to a declaration of a constant:

```
consts id :: "type_expr"
```

where `type_expr` is the result of translating type_expr as previously described.

3.5 Translating Axiom Declarations

An axiom declaration consists of a named equivalence expression:

axiom
[axiom_id] value_expr_1 ≡ value_expr_2

This is translated to:

```
axioms
  axiom_id : "value_expr_1 = value_expr_2"
```

where the value expressions have been translated accordingly as described in the next section.

Since RSL value expressions are represented in Isabelle/HOL as terms expressing the semantics of the value expressions, and the semantics of ≡ amounts to equality of the semantics of the value expressions, object-level equality is used to represent ≡ in Isabelle/HOL.

3.6 Translating Value Expressions

Value Literals The values of the basic RSL types are denoted by value literals. The boolean RSL literal **true** and **false** are translated to terms `true` and `false` defined by:

```
constdefs
  true :: "bool Process"
  "true == Proc True"

  false :: "bool Process"
  "false == Proc False"
```

These constants express the semantics of the literals.

The unit literal () is translated to `Proc ()`, and integer literals are translated such that e.g. **12** becomes `Proc #12`.

Value Names Names of user-declared values may occur in value expressions. Since a value name represents a *value* and the occurence of a value name in a value expression must represent a *process*, a value name `id` is represented in a HOL term as `(Proc id)`.

Basic Expressions The basic expressions **chaos** and **skip** are translated to terms `chaos` and `skip` defined by:

```
constdefs
  chaos :: "'a Process"
  "chaos == Chaos"

  skip :: "unit Process"
  "skip == Proc ()"
```

Infix Expressions Infix expressions are categorized as axiom infix expressions and value infix expressions. Axiom infix expressions are built from boolean expressions and the logical connectives $\wedge$, $\vee$, and $\Rightarrow$. Value infix expressions are, e.g., built from integer expressions and arithmetic operators.

Infix expressions are represented by HOL terms built using infix operators. As an example, an axiom infix expression of the form:

boolean_value_expr_1 $\wedge$ boolean_value_expr_2

is translated to:

```
boolean_value_expr_1 AND boolean_value_expr_2
```

where `AND` is infix syntax for `RSL_and` defined by:

```
constdefs
 RSL_and :: "(bool Process) => (bool Process) => (bool Process)"
 "bp1 AND bp2 == infix_c_appl bp1 RSLand bp2"
```

where `infix_c_appl` merely pipes the value of `bp1` to `RSLand` defined by:

```
constdefs
 RSLand :: "bool => (bool Process) => (bool Process)"
 "RSLand b p == if b then p else (v2p False)"
```

The definition of `infix_c_appl` is:

```
constdefs
 infix_c_appl :: "(bool Process) => (bool => (bool Process) =>
    (bool Process)) => (bool Process) => (bool Process)"
 "infix_c_appl ve1 iconn ve2 == ve1 pipe (% dv1 . iconn dv1 ve2)"
```

This representation of a conjunction is derived from the denotational semantics of RSL where the semantics of an infix expression with a logical connective in the context of a signature Σ and a model m is given by:

M(value_expr_1 infix_connective value_expr_2)(Σ)(m) =
  M(value_expr_1)(Σ)(m) pipe
    (λ dv1 $\in$ BooleanValues •
      M(infix_connective)(dv1)(M(value_expr_2)(Σ)(m))
)

`RSLand` corresponds directly to the semantics of the connective for conjunction given by:

M($\wedge$)(b)(p) = **if** (b = True) **then** p **else** v2p(False) **end**

where b is a boolean value and p is a process returning a boolean value (if terminating).

Disjunctions and implications are handled the same way. In Sect. 4, these translations are used when representing RSL proof rules in Isabelle/HOL.

Value infix operators are translated similarly, and their representations rely on their counterparts found in Isabelle/HOL. The operators are not specified in detail in the denotational semantics [7]; but an informal description is found in [11]. Since the operators of Isabelle/HOL conform with the descriptions in [11], they are used to represent the RSL operators.

Prefix Expressions Prefix expressions are categorized as axiom prefix expressions and value prefix expressions, and the translation of prefix expressions is similar to that of infix expressions.

There is one prefix connective: logical negation ($\sim$). And as for value infix expressions, the semantics of the prefix operators is not specified any further in [7], but it is described informally in [11].

Quantified Expressions A universally quantified expression of the form:

($\forall$ id : type_expr • boolean_value_expr)

is translated to:

```
v2p (ALL id :: type_expr . (boolean_value_expr) = true)
```

where type_expr and boolean_value_expr have been translated accordingly. Occurences of id in boolean_value_expr are translated to `(Proc id)`.

Since `ALL` is the universal quantifier in Isabelle/HOL, the body must be of type `bool`. The comparison with `true` is used to bring the body of type `bool Process` to type `bool`. This also handles divergence (i.e. `Chaos`) correctly.

The above translation is not a direct representation of the RSL semantics in which quantified expressions are treated by comparing sets. For a universally quantified expression of the form ($\forall$ x : T • P(x)), let two sets, V1 and V2, be defined by:

V1 = { v | type_of(v) = T }
V2 = { v | v $\in$ V1 $\wedge$ P(v) }

The semantics of the quantified expression is then:

vp2 (V1 ∩ V2 = V1)

This could be expressed directly in Isabelle/HOL as:

```
v2p ((% V1 V2 . (V1 Int V2 = V1))
     (Collect (% x :: 'a . True))
     (Collect (% y :: 'a . y : (Collect (% x :: 'a . True)) &
                               ((P (Proc y)) = Proc True)))
    )
```

However, this is not very readable and does not utilize the quantifiers of Isabelle/HOL. Consequently, we have chosen to translate as described above after we used Isabelle to prove that the direct representation of the semantics is equivalent with:

```
v2p (ALL i :: 'a . (P (Proc i)) = true)
```

Existential quantification is handled in a similar way.

Structured Expressions We limit our translation to if-expressions. A constant for expressing an if-expression is defined as follows:

```
constdefs
ifexpr :: "bool Process => 'a Process => 'a Process => 'a Process"
"if veb then ve1 else ve2 end ==
 veb pipe (%dv1. if dv1 then (ve1 pipe v2p) else (ve2 pipe v2p))"
```

It follows the RSL semantics and translating an if-expression to an application of this constant gives us a more readable result of the translation. However, when reasoning about the translated if-expression, it must be unfolded using the rule `ifexpr_def`.

Function Expressions A function expression of the form:

λ id : type_expr • value_expr

is translated to:

```
v2p (% id :: type_expr . value_expr)
```

where occurences of id in value_expr are translated to `(Proc id)`.

Function Application Expressions An operator for function application is introduced:

```
constdefs
appl :: "('a => 'b Process) Process => ('a Process) => ('b Process)"
"appl f x == f pipe (% dv1 . x pipe (% dv2 . dv1 dv2))
```

The right-hand side of the defining equation corresponds to the semantics of function applications. As a consequence of the use of the pipe operator, function application is strict.

3.7 The RSL Proof Rules as Theorems

RSL not only has a denotational semantics but also a large set of proof rules. Some of these proof rules are translated into Isabelle/HOL and proved as theorems. These theorems ease the task of proving properties of translated specifications; and since the translation may be viewed as an encoding of the denotational semantics, proving proof rules as theorems also shows that the proof rules are sound with respect to the denotational semantics.

There are different kinds of RSL proof rules: equivalence rules and inference rules. Both kinds may have side-conditions expressed by predicates such as **convergent** and **readonly**. Side-conditions regarding imperative variables and channels are not relevant in this simple setting, but convergence must be dealt with. If an expression eb is required to be convergent in an RSL proof rule, we let it occur as `(Proc eb)` in the corresponding Isabelle/HOL theorem. As an example, the RSL proof rule:

$$[\,\text{true\_or\_false}\,] \quad (\text{eb} = \textbf{true}) \vee (\text{eb} = \textbf{false}) \simeq \textbf{true}$$

(when eb is convergent and readonly) is expressed as:

```
theorem true_or_false :
  "((Proc eb) =rsl true) OR ((Proc eb) =rsl false) == true"
```

The theorem is easily proved using `eq_reflection` and simplification with the definitions of the relevant operators.

Inference rules are expressed as meta-level implications in Isabelle.

We have encoded a series of relevant proof rules as theorems and proved them in Isabelle/HOL. Most of the proofs could easily be carried out by simplification and case analysis (divergence and defined result).

4 Example Translation

In this section, we present an example in which an RSL specification is translated to Isabelle/HOL and properties are proved using Isabelle.

4.1 RSL Specification

The following RSL scheme specifies a function xor:

scheme XOR =
  class
    type Signal = **Bool**
    value xor : Signal $\xrightarrow{\sim}$ Signal $\xrightarrow{\sim}$ Signal
    axiom
      [xor_def]
        xor $\equiv$ (λ s1 : Signal • (λ s2 : Signal • ($\sim$s1 $\wedge$ s2) $\vee$ (s1 $\wedge$ $\sim$s2)))
  end

Applying the function xor to two signals (booleans) yields true if and only if the signals are different and none of them equate chaos. Recall that function application is strict.

4.2 Result of Translation

The result of translating the scheme XOR is an Isabelle theory:

```
theory XOR = RSL :
types Signal = bool
consts xor :: "Signal => ((Signal => ((Signal) Process)) Process)"
axioms xor_def : "((Proc xor)) = (v2p (% s1 :: Signal .
 (v2p (% s2 :: Signal . ((((NOT ((Proc s1))) AND ((Proc s2)) OR
 (((Proc s1)) AND (NOT ((Proc s2))))))))))"
end
```

4.3 Proving Properties

We have used Isabelle to prove a number of properties of the xor function. As an example, it is commutative, expressed by:

$$(\forall\ \mathrm{s1} : \mathrm{Signal} \bullet (\forall\ \mathrm{s2} : \mathrm{Signal} \bullet (\mathrm{xor(s1)})(\mathrm{s2}) = (\mathrm{xor(s2)})(\mathrm{s1}))) \equiv \mathbf{true}$$

This is translated to a lemma:

```
lemma xor_com : "(v2p (ALL s1 :: Signal . ((v2p (ALL s2 :: Signal
  . ((((appl (appl (Proc xor) (Proc s1)) (Proc s2))) =rsl
  ((appl (appl (Proc xor) (Proc s2)) (Proc s1)))) = true))) =
  true))) = (true)"
```

Although it looks complicated, it is easily proved by simplification with a set of rules including `xor_def` and relevant definitions (i.e. `appl_def`, `true_def`, etc.).

We also verified that the result of applying xor to a divergent signal is divergent, and that the function behaves as expected otherwise. Although the strictness of xor is caused by function application, it is consistent with the results of simply replacing xor(s1)(s2) with $(\sim\mathrm{s1} \wedge \mathrm{s2}) \vee (\mathrm{s1} \wedge \sim\mathrm{s2})$.

4.4 Lessons Learned

In RSL specifications, universal quantifiers are often used in axioms that specify functions. As an example, an axiom specifying xor could have been:

axiom
[xor_def]
$(\forall\ \mathrm{s1} : \mathrm{Signal} \bullet (\forall\ \mathrm{s2} : \mathrm{Signal} \bullet \mathrm{xor(s1)(s2)} \equiv (\sim\mathrm{s1} \wedge \mathrm{s2}) \vee (\mathrm{s1} \wedge \sim\mathrm{s2})))$

However, this formulation is not suited for rewriting the term `(Proc xor)`, and it turned out that the formulation using lambda abstractions was preferable when the translated axiom was to be used by the simplifier.

5 Expanding the Translated Subset of RSL

We will now discuss various possibilities for expanding the subset of RSL which is translated.

5.1 Explicit Function Definitions

In RSL, functions are often declared by explicit function definitions, e.g.:

value
xor : Signal $\xrightarrow{\sim}$ Signal $\xrightarrow{\sim}$ Signal
xor(s1)(s2) ≡ (∼s1 ∧ s2) ∨ (s1 ∧ ∼s2)

Conceptually, this is a short way of writing a value declaration and a defining axiom, and it is tempting to translate it using `constdefs` which combines declaration and definition of a constant in Isabelle. However, this does not go well with our use of an operator for function application.

Based on our experience with the example in Sect. 4, a possible approach is to generate an axiom with lambda abstractions as in the example.

5.2 Imperative Variables

Full RSL offers imperative variables. Variables are declared with a name and a type, and they may occur in value expressions. The value of a variable is changed by assignment expressions of the form x ::= *ve* which has the unit value. The side-effect of the assignment is that the value of the variable x is set to *ve*.

To give semantics to value expressions with imperative variables, the notion of a store is introduced. A *store* is a function that maps names of variables to their values. The semantics of a value expression is now an *effect* which is a function that maps a store to a "process". A "process" is now either (*i*) a store and a value or (*ii*) the divergent process. A pair consisting of a store and a value is also known as a *routine*, and an *action* is a function that maps a value to an effect.

Types for routines, processes, effects, and actions are declared as follows:

```
types 'a Routine = "Store * 'a"
datatype 'a Process_ = Proc 'a | Chaos
types 'a Process = "('a Routine) Process_"
types 'a Effect = "Store => ('a Process)"
types ('a,'b) Action = "'a => ('b Effect)"
```

If we also declare a type `Vars` for representing names of variables and only consider integer variables, we can represent a store as a function of the type `Vars => int`.

The semantic operators are then redefined to accomodate piping of processes and effects, and definitions for RSL constructs (e.g. basic expressions)

are changed accordingly such that effects are considered instead of processes. This is done systematically without problems.

The problems arise when proof rules are attempted to be proved as theorems. The theorems expressing proof rules are easily modified, but the proof scripts cannot be rerun since the case analysis based on the old datatype for processes is no longer available and functions must be considered. Consequently, the proofs become difficult.

5.3 Non-determinism and Concurrency

To give semantics to non-deterministic value expressions in RSL, the concept of a terminating process is extended such that a process has a set of possible result values. Although this may be expressed directly in Isabelle/HOL, reasoning about non-deterministic expressions is expected to become tedious.

Including concurrency and communication on channnels also increases the complexity of the semantics. A process then has a state determined by the communications it may perform, and a set of possible states must be added to the representation of a process. This is expected to make the result of a translated RSL expression hard to read and reason about.

6 Conclusion

6.1 Achievements

We have presented a representation of an applicative, deterministic subset of RSL in Isabelle/HOL, making it possible to translate basic RSL specifications to Isabelle/HOL theories and then use Isabelle.

RSL proof rules have been translated into Isabelle/HOL theorems and proved using Isabelle. Since the translation/representation corresponds closely to the denotational semantics, proving RSL proof rules as theorems not only provides helpful theorems for future proofs but also proves the RSL proof rules sound with respect to the denotational semantics. Such a formal proof of soundness is a novelty.

Various possibilities for expanding the translated subset of RSL were discussed. This may be done by following the approach of translating by encoding the semantics. However, as the subset expands, the semantics becomes more complicated and so does the result of a translation.

6.2 Related Work

Institutions are used as the theoretical foundation for our translation, and this approach has also been used to reuse Isabelle/HOL for the specification language CASL [1, 8].

A formal semantics of OCL embedded into Isabelle/HOL is presented in [2] and used to derive proof calculi by proving the proof rules with Isabelle. Thereby the calculi are consistent and sound with respect to the semantics. Essentially, we did the same thing when proving the RSL proof rules as theorems.

6.3 Future Work

Immediate future perspectives are to expand the translated subset of RSL, prove more RSL proof rules as theorems, and perform realistic case-studies. Some RSL constructs may be expressed by the subset already translated, e.g., the explicit function definitions. Furthermore, translating structured specifications may also be considered.

Another perspective is to provide XEmacs support for the translation such that it can be used easily together with the ProofGeneral user-interface for proof assistants which works well as an interface for Isabelle.

Acknowledgements

The author would like to thank Anne Haxthausen for comments on drafts of this paper and Henrik Pilegaard for inspirational discussions on using Isabelle.

References

1. Borzyszkowski, T.: Higher-Order Logic and Theorem Proving for Structured Specifications. In Choppy, C., Bert, D., Mosses, P. (eds.): *Recent Trends in Algebraic Development Techniques, Selected Papers, 14th International Workshop WADT'99*. LNCS 1827, pp. 401-418. Springer, 1999.
2. Brucker, A.D, Wolff, B.: A Proposal for a Formal OCL Semantics in Isabelle/HOL. In Carreño, V.A., Muñoz, C., Tahar, S. (eds.): *TPHOLs 2002*. LNCS 2410, pp. 99-114. Springer, 2002.
3. Bruun, P.M., Dandanell, B., Gørtz, J., Haff, P., Heilmann, S., Prehn, S., Haastrup, P., Reher, J., Snog, H., Zierau, E.: *RAISE Tools Reference Manual*. LACOS/CRI/DOC13/0/V2. 1993.
4. George, C.: *RAISE Tools User Guide*. Technical Report 227, UNU/IIST, 2001.
5. Gordon, M.J.C., Melham, T.F. (eds.): *Introduction to HOL — A theorem proving environment for higher order Logic*. Cambridge University Press, 1993.
6. Lindegaard, M.P., Haxthausen, A.E.: Using Institutions to Provide Proof Support for RSL. *To be submitted*.
7. Milne, R.: *Semantic Foundations of RSL*. RAISE/CRI/DOC/4/V1, 1990.
8. Mossakowski, T.: *Relating CASL with Other Specification Languages: the Institution Level*. Theoretical Computer Science 286, pp. 367-475. 2002.
9. Nipkow, T., Paulson, L.C., Wenzel, M.: *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.
10. Paulson, L.C.: *Isabelle – A Generic Theorem Prover*. LNCS 828. Springer, 1994.
11. The RAISE Language Group: *The RAISE Specification Language*. The BCS Practitioners Series. Prentice Hall Int., 1992.
12. The RAISE Method Group: *The RAISE Development Method*. The BCS Practitioners Series. Prentice Hall Int., 1995.
13. Tarlecki, A.: Institutions: An Abstract Framework for Formal Specifications. In Astesiano, E., Kreowski, H.-J., Krieg-Brückner, B. (eds.): *Algebraic Foundations of Systems Specification*. IFIP state-of-the-art report. Springer, 1999.

Verifying Functional Bulk Synchronous Parallel Programs Using the Coq System

Frédéric Gava and Frédéric Loulergue

Laboratory of Algorithms, Complexity and Logic
University Paris XII, Val-de-Marne
61, avenue du Général de Gaulle
94010 Créteil cedex – France
Tel: +33 (0)1 45 17 16 50
Fax: +33 (0)1 45 17 66 01
{gava,loulergue}@univ-paris12.fr

Abstract. The Bulk Synchronous Parallel ML (BSML) is a functional language for Bulk Synchronous Parallel (BSP) programming. It is based on an extension of the λ-calculus with parallel operations on a parallel data structure named parallel vector, which is given by intention. We present the formal proofs of correctness for BSML programs in the Coq proof assistant. Such developments demonstrate the usefulness of higher-order logic in the process of software certification and parallel applications. They also show that proof of rather complex parallel algorithms may be made with inductive types by using existing certified programs.
Keywords: Parallel Programming, Bulk Synchronous Parallelism, Functional Programming, Certification, Coq, Theorem prover

1 Introduction

Some problems require performance that can only be provided by massively parallel computers. Programming this kind of computers is still difficult. Works on functional programming and parallelism can be divided in two categories: explicit parallel extensions of functional languages — where languages are either non-deterministic [40] or non-functional [2, 16] — and parallel implementations with functional semantics [1] — where resulting languages do not express parallel algorithms directly and do not allow the prediction of execution times. Algorithmic skeleton languages [9, 11, 43, 7], in which only a finite set of operations (the skeletons) are parallel, constitute an intermediate approach. Their functional semantics is explicit but, their parallel operational semantics is implicit. The set of algorithmic skeletons has to be as complete as possible, but it often depends on the domain of application.

The design of parallel programming languages is a tradeoff between:

- the possibility to express the parallel features that are necessary for predictable efficiency, but with programs that are more difficult to write, prove and port

– the abstraction of such features that are necessary to make parallel programming easier, but which must not hinder efficiency and performance prediction.

We are exploring thoroughly the intermediate position of the paradigm of algorithmic skeletons in order to obtain universal parallel languages where execution cost can easily be determined from the source code (in this context, cost means the estimate of parallel execution time). This last requirement forces the use of explicit processes corresponding to the processors of the parallel machine. *Bulk Synchronous Parallel* (BSP) computing [37, 45] is a parallel programming model which uses explicit processes, offers a high degree of abstraction and yet, allows *portable* and *predictable* performance on a wide variety of architectures.

An operational approach has led to a BSP λ-calculus that is confluent and universal for BSP algorithms [36], and to a library of bulk synchronous primitives for the Objective Caml [30] language which is sufficiently expressive and makes the prediction of execution times [24, 34] possible.

This framework is a good tradeoff for parallel programming because:

– we defined a *confluent calculus* so:
  - we can design purely functional parallel languages from it. Without side-effects, programs are easier to prove, and to re-use (the semantics is compositional)
  - we can choose any evaluation strategy for the language. An eager language allows good performances.
– this calculus is based on BSP operations, so programs are easy to port, their costs can be predicted and are also portable because they are parametrized by the BSP parameters of the target architecture.

Bulk Synchronous Parallel ML or BSML is our extension of ML for programming direct-mode parallel BSP algorithms as functional programs. A BSP algorithm is said to be in *direct mode* [20] when its physical process structure is made explicit. Such algorithms offer predictable and scalable performance and BSML expresses them with a small set of primitives taken from the *confluent* BSλ-calculus [36] : a parallel constructor, asynchronous parallel function application, synchronous global communications and a synchronous global conditional.

There is currently no full implementation of BSML but there is a partial implementation as a library. The `BSMLlib` library implements the BSML primitives using Objective Caml [30] and MPI [47]. With its small number of basic operations, `BSMLlib` can be taught to BSc. students (universities of Orléans and Paris Val de Marne). There are additional modules which provide several usual parallel algorithms. They constitute what is called the `BSMLlib` standard library. In addition, its performance follows curves predicted by the BSP cost model [3]. This environment is a safe one. Our language is deterministic and is based on a parallel abstract machine [39] which has proven correct w.r.t. the confluent BSλ_p-calculus [31] using an intermediate semantics [32]. A polymorphic type system [19] has been designed, for which type inference is possible.

We are now interested in the *certification* of BSML programs. This paper describes our first work in this direction. Section 2 presents functional bulk synchronous parallel programming. The formalization of the BSML operators in Coq is given in section 3. It is used for verifying properties of several BSML programs which are part of the standard library of `BSMLlib` (section 4). We end with related work (section 5) and future work (section 6).

2 Functional Bulk Synchronous Parallelism

2.1 Bulk Synchronous Parallelism

The Bulk Synchronous Parallel (BSP) model [53, 38, 45] describes an abstract parallel computer, an execution model, and a cost model. A BSP computer has three components: a homogeneous set of processor-memory pairs, a communication network for inter processor delivery of messages and a global synchronization unit which executes collective requests for a *synchronization barrier*. A wide range of actual architectures can be seen as BSP computers.

The performance of the BSP computer is characterized by three parameters (expressed as multiples of the local processing speed): the number of processor-memory pairs **p** ; the time **l** required for a global synchronization ; the time **g** for collectively delivering a 1-relation (communication phase where every processor receives/sends at most one word). The network can deliver an h-relation (communication phase where every processor receives/sends at most h words) in time $g \times h$. These parameters can easily be obtained using benchmarks [27].

A BSP program is executed as a sequence of *super-steps*, each one divided into (at most) three successive and logically disjointed phases (Fig. 1):

1. Each processor uses its local data (only) to perform sequential computations and to request data transfers to/from other nodes;
2. the network delivers the requested data transfers;
3. a global synchronization barrier occurs, making the transferred data available for the next super-step.

Bulk Synchronous Parallelism (and the Coarse-Grained Multicomputer, CGM, which can be seen as a special case of the BSP model) is used for a large variety of applications: scientific computing [5, 28], genetic algorithms [8] and genetic programming [12], neural networks [44], parallel databases [4], constraint solvers [21], *etc.* As stated in [13] "A comparison of the proceedings of the eminent conference in the field, the ACM Symposium on Parallel Algorithms and Architectures, between the late eighties and the time from the mid nineties to today reveals a startling change in research focus. Today, the majority of research in parallel algorithms is within the coarse-grained, BSP style, domain".

2.2 Bulk Synchronous Parallel ML

There is currently no implementation of a full Bulk Synchronous Parallel ML language but rather a partial implementation: a library for Objective Caml. The so-called `BSMLlib` library is based on the following elements.

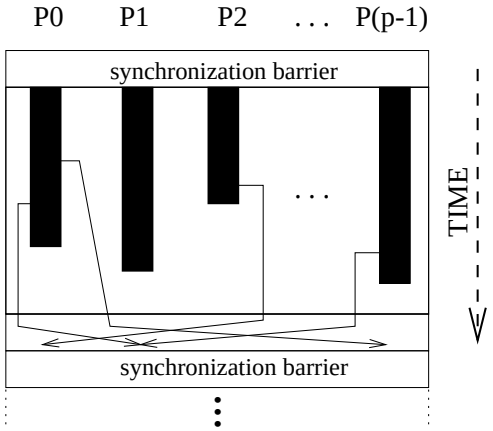


Fig. 1. A BSP super-step

It gives access to the BSP parameters of the underling architecture. In particular, it offers the function `bsp_p:unit->int` such that the value of `bsp_p()` is p, the static number of processes of the parallel machine. The value of this variable does not change during execution (for "flat" programming, this is not true if a parallel juxtaposition is added to the language [33]).

There is also an abstract polymorphic type `'a par` which represents the type of p-wide parallel vectors of objects of type `'a`, one per process. The nesting of `par` types is prohibited. Our type system enforces this restriction [19]. This improves on the earlier design DPML/Caml Flight [23, 16] in which the global parallel control structure `sync` had to be prevented *dynamically* from nesting.

This is very different from SPMD programming (Single Program Multiple Data) where the programmer must use a sequential language and a communication library (like MPI [47]). A parallel program is then the multiple copies of a sequential program, which exchange messages using the communication library. In this case, messages and processes are explicit, but programs may be *non deterministic* or may contain *deadlocks*.

The parallel constructs of BSML operate on parallel vectors. Those parallel vectors are created by:

```
mkpar: (int -> 'a) -> 'a par
```

so that `(mkpar f)` stores `(f i)` on process i for i between 0 and $(p-1)$. We usually write `f` as `fun pid->e` to show that the expression `e` may be different on each processor. This expression `e` is said to be *local*. The expression `(mkpar f)` is a parallel object and it is said to be *global*.

A BSP algorithm is expressed as a combination of asynchronous local computations (first phase of a super-step) and phases of global communication (second phase of a super-step) with global synchronization (third phase of a super-step). Asynchronous phases are programmed with `mkpar` and with:

```
apply: ('a -> 'b) par -> 'a par -> 'b par
```

`apply (mkpar f) (mkpar e)` stores `(f i) (e i)` on process i. Neither the implementation of `BSMLlib`, nor its semantics [32] prescribe a synchronization barrier between two successive uses of `apply`.

Readers familiar with BSPlib [45, 27] will observe that we ignore the distinction between a communication request and its realization at the barrier. The communication and synchronization phases are expressed by:

```
put:(int->'a option) par -> (int->'a option) par
```

Consider the expression: `put(mkpar(fun i->fs`$_i$`))` (*)

To send a value `v` from process `j` to process `i`, the function `fs`$_j$ at process `j` must be such as (`fs`$_j$ `i`) evaluates to `Some v`. To send no value from process `j` to process `i`, (`fs`$_j$ `i`) must evaluate to `None`.

Expression (*) evaluates to a parallel vector containing a function `fd`$_i$ of delivered messages on every process. At process `i`, (`fd`$_i$ `j`) evaluates to `None` if process `j` sent no message to process `i` or evaluates to `Some v` if process `j` sent the value `v` to the process `i`.

The full language would also contain a synchronous conditional operation:

```
ifat: (bool par) * int * 'a * 'a -> 'a
```

such that `ifat (v,i,v1,v2)` will evaluate to `v1` or `v2` depending on the value of `v` at process `i`. But Objective Caml is an eager language and this synchronous conditional operation can not be defined as a function. That is why the core `BSMLlib` contains the function: `at:bool par -> int -> bool` to be used only in the construction: `if (at vec pid) then...else...` where `(vec:bool par)` and `(pid:int)`. `if at` expresses communication and synchronization phases. Global conditional is necessary to express algorithms like :

Repeat Parallel Iteration **Until** Max of local errors < **epsilon**

Without it, the global control cannot take into account data computed locally.

2.3 Advantages of Functional BSP Programming

The functional approach of this parallel model allows the re-use of suitable technical for formal proof from functional languages because a few numbers of parallel operators is needed to an explicit parallel extension of a functional language. Those operators (for a static number of processes) of the BSML language are derived from a confluent calculus [36] so parallel algorithms are also confluent and keep the advantages of the BSP models: no deadlock, efficient implementation using optimized communication algorithms, static cost formulae and cost previsions. A powerful axiomatization of theses parallel operators enables to express all BSP algorithms in a natural way. Thus, we could *prove* and *certify* functional implementation of those algorithms. The *extraction* possibility of proof assistant can be used to generate a *certified library* of *parallel algorithms* to be used independently of the sequential or parallel implementation of BSML operators [34].

3 Formalization of `BSMLlib` in Coq

The **Coq** System [51] is a proof assistant for high-order logic based on the Calculus of Inductive Construction [10], a typed λ-calculus, extended with *inductive* definitions. It is a language of terms and types where terms can be seen as representing proofs or programs and types as representing *specifications* of data types. It makes it possible to write specifications and *propositions*, to check mathematical proofs, and to synthesize computer programs from proofs of their specifications. Indeed, there is a *constructive interpretation* of proofs, i.e. proving a formula implies explicitly constructing a typed λ-term. This suggests a uniform framework for representing formulas, proving programs and identifying those notions [41]. Via the Curry-Howard isomorphism, a proof of a logical formula $\forall x.P(x) \Rightarrow \exists y.Q(x,y)$ (called a specification) needs to be derived in order to procude a *correct* program in such a formalism. The program p, as $y = (p\ x)$, is obtained by *extraction* from the proof of the formula by forgetting everything concerning the logical correctness of the program; in particular, proofs of logical assertions and intermediate properties of sub-programs [42]. As a typed λ-calculus, the logic of the **Coq** system is naturally well-suited to prove purely functional programs [41]. We show that is also possible to certify the correctness of parallel programs.

To represent our parallel language and have an specification-extraction of functional BSP programs, we choose a classical approach: an *axiomatization* of the parallel operators which are based on the parallel programming model of the BSλ-calculus. Thus, operations are given by *parameters* and do not depend on the implementation (sequential or parallel). The formalization in **Coq** is based on the `BSMLlib` elements: the number of processes, the parallel vectors and their operators.

The number of processes is naturally a non negative integer given by the parameters. Parallel vectors are indexed over type Z (Coq's integers), starting from 0 to the constant number of processes. They are represented in the logical world by an abstract dependent type `Vector T` where `T` is the type of its elements. This abstract type is only manipulated by one of the parameters. We give for example, the `mkpar` operator:

```
Parameters bsp_p:unit -> Z;
           Vector : Set -> Set;
           mkpar: (T:Set) (Z->T) -> (Vector T).

Axiom good_bsp_p:‘0 < (bsp_p tt)‘.

Axiom at: (T:Set) (Vector T) -> (i:Z) ‘0<=i<(bsp_p tt)‘ -> T.
```

`at` is an abstract "access" function for the parallel vectors. It gives the local value contained at a processor and it would be used in the specification of programs to give the values contained in the parallel vector result. It has a dependent type to verify that the number `i` is a valid number of process. The operators for

local computation are the constructor `mkpar` of parallel vectors and the global application. They are axiomatized using the Coq's syntax:

```
Axiom mkpar_def: (T:Set) (f:Z -> T)(i:Z) (H:‘0 <= i < (bsp_p tt)‘)
                   (at T (mkpar T f) i H)=(f i).

Axiom apply_def: (T1,T2:Set) (V1:(Vector (T1 -> T2)))
                 (V2:(Vector T1)) (i:Z) (H:‘0 <= i < (bsp_p tt)‘)
(at T2(apply T1 T2 V1 V2)i H)=((at(T1->T2) V1 i H)(at T1 V2 i H)).
```

For a function `f`, `mkpar_def` stores `(f i)` on the process `i` by using the access function `at` and in the same manner `apply_def` stores `(V1 i) (V2 i)`. The parallel vector result is described with an equality which has the type `Prop`. The result is given for a parameter `i` which has to be proved as a valid number of process.

The communication of values [36] has been first a `get` operator: each process gets a unique value from another process. Its implementation and the associated axioms are easy to write. But for some algorithms (essentially optimized BSP algorithms), the dual operator, `put`, is necessary. The above two functions are axiomatized as follows:

```
Axiom get_def: (T:Set)(V1:(Vector T))(V2: (Vector Z))
               (i:Z) (H_i:‘0 <= i < nprocs‘) H_vec:((j:Z)
(H_j:‘0 <= j < (bsp_p tt)‘) -> ‘0<= (at V2 j H_j) < (bsp_p tt)‘)
    -> (at (get V1 V2) i H_i)=(at V1 (at V2 i H_i) (H_vec i H_i)).

Axiom put_def: (T:Set)(Vf:(Vector (Z -> (option T))))
               (i:Z) (H:‘0 <= i < (bsp_p tt)‘)
((at (Z->(option T)) (put T Vf) i H)=([j:Z] if (within_bound j)
    then [H1]((at (Z-> (option T)) Vf j H1) i)
     else [H2](None T))).
```

It transforms a functional vector to another functional vector which guides communication using the functional parameter `j` to read values from distant processes. The parameter `j` is tested with the function `within_bound` of type:

```
(i:Z) {‘0<=i<(bsp_p tt)‘}+{~‘0<=i<(bsp_p tt)‘}
```

which tells whether an integer is a valid number of process or not and which gives the proof. If it does, the value on the process `i` is read on process `j` with the access function (`j` is a valid number of process; the proof is given by `within_bound`). Otherwise, an empty constant is returned. In a real implementation, the values to emit are first calculated and after exchanged with an optimal algorithm [34].

The full axiomatization, would also contains the synchronous global conditional. Its axiomatization is easy to express using the `Bool` libraries of the **Coq** system:

```
Axiom ifat_def: (T:Set)(V:(Vector bool)) (n:Z)(R_IF,R_ELSE:T)
(Hyp_n: ‘0 <= n < (bsp_p tt)‘) ((ifat T V n R_IF R_ELSE)=
        (if (sumbool_of_bool (at bool V n Hyp_n)) then [H1] R_IF
           else [H2] R_ELSE)).
```

`sumbool_of_bool` is a function which transforms a boolean to the proof that is true or not. For proofs of programs, it is easier to directly manipulate proofs than primitive constants. The parameter `n` needs to be a valid number of process and this proof is given to the "access" function by a dependent type. Since `ifat` is a function, in this axiomatization, expressions using a global conditional would be also extracted by the **Coq** system as a function. Thus, the extracted code needs to be parsed in order to be transformed into a suitable conditional. Now, with the set of axioms of our parallel operators, we will be able to verify the formal properties of classical functional BSP programs.

4 Formal Proofs of BSML Programs

We present here the proof of some parallel algorithms in the **Coq** system. In order to simplify the presentation and to ease the formal reasoning, we limit our study and formal proofs to the expressions which are very common in a BSP algorithm and the two different ways to communicate values in BSML. Those case studies have demonstrated the relevance of the use of the **Coq** system in the proof of parallel programs *correctness*. In particular, we used defined predicates and libraries several times in the developments and we also used higher axioms to define a new principle and to prove correctness of programs [41] and in our case of BSML programs.

To increase the readability of this section, we give the expressions in a Objective Caml [30] syntax given by the **Coq** system program extraction [42] (with minimal manual modification). The formal developments described in this section are freely available [17]. The **Coq** system and the tactics used to prove the correctness of our programs are also freely available and described at the web page of the **Coq** System [51].

Replicate The first case study we present is one of the simple functional BSP expressions: the *replication* of the same value on each process. It is often used at the beginning of BSP algorithms to replicate all the parameters. As we explained before, the specification of `replicate` could be read as the following formala:

$$\forall \text{Data}\ \forall a : \text{Data} \Rightarrow \exists res\ \text{as}\ \forall i\ (0 \leq i < p)\ (\text{res}[i] = a)$$

where $res[i]$ is an abbreviation for the function access of local data (we do not write the type of an element when it is intuitive). For this trivial example, we explain the realization in **Coq**. First, we have to introduce the hypothesis in the context with the tactic `Intros`. Then, we give the desired parallel vector with `Exists (mkpar T ([pid:Z]a))`, prove that it satisfies the specification using the `mkpar_def` axiom and by this means, **Coq** could finish automatically (`Rewrite mkpar_def;Auto`). The extracted program will fit the desired one:

```
let replicate param = mkpar (fun x -> param);;
val replicate : 'a -> 'a par
```

It is a property which will be always used in the following.

Compositionality To validate our study, we need to give formal proof of the core library. For example, we give here a definition [33] of a weak form of parallel composition analogous to a data-parallel conditional *where* statement to cover the whole network.

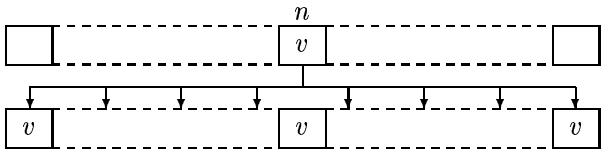
```
let mask c x y = apply (apply (mkpar (fun j x0 y0 ->
                            if (c j) then x0 else y0)) x ) y;;

val mask : (int -> bool) -> 'a par -> 'a par -> 'a par
```

This expression has been used in [36], to prove the capacity of the equational theory of the BSλ-calculus and its functional implementation. But without a proof assistant, the authors have made some bad applications of the axioms (Lemma 4, `mask` commutes with `apply`):

```
Lemma mask_is_composition_apply:(T:Set)(c:(Z->bool))(f,g:Z->T->T)
                          (f',g':Z->T)(i:Z)(H_i:`0<=i<(bsp_p tt)`)
(at (mask c (apply (mkpar f) (mkpar f'))
            (apply (mkpar g) (mkpar g'))) i H_i)
= (at (apply (mask c (mkpar f) (mkpar g))
             (mask c (mkpar f') (mkpar g'))) i H_i).
```

which have been proved by the Coq system by trivial application of the axioms without forgetting cases. The necessity to certify the properties of our programs will appear in the following part of this section.

Broadcast of a value *Exchange* of values is the critical point of parallel algorithms. BSML offers two different operators for this work: `get` and `put` which are complementary. We express this by a classical example: the *broadcast* of an element v held by a process n:



To limit its use and to avoid the failure of the program, n needs to be a valid number of process. If not, we modelize this exception by returning the `False` proofs. Thus, the fully specified function should have the following type:

$\forall$Data $\forall n\ (0 \leq n < p)\ \forall vv \Rightarrow \exists res$ as $\forall i(\ 0 \leq i < p)\ (\mathrm{res}[i] = \mathrm{vv}[n])$.

Broadcasting can be done in BSML, using the `get` operator, with the following function:

```
let broadcast_get n vect = get vect (replicate n);;
val broadcast_get : int -> 'a par -> 'a option par
```

We prove this property by cases on n. Thus, to broadcast a value from a process n, we need that n is really the *pid* of a process. If not, we cannot prove our algorithm and like in the implementation, the program fails.

Another way to express this problem, is using the `put` operation for broadcasting a value. With this new definition of the broadcast, we can prove an interesting result, which performs and modelises error of the broadcast by having `None` on each process:

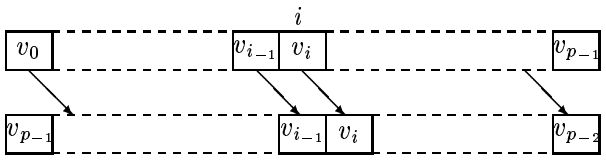
$$\forall \text{Data}\ \forall n\ \forall vv : (Vector\ Data) \Rightarrow$$
$$(\exists res \text{ as } (\ 0 \leq n < p)\ \forall i(\ 0 \leq i < p)\ (\text{res}[i] = Some(\text{vv}[n])))$$
$$\vee$$
$$(\exists res \text{ as } not(\ 0 \leq n < p)\ \forall i(\ 0 \leq i < p)\ (\text{res}[i] = None)$$

and we prove it by cases on n and by contradiction. The following expression is the realizer:

```
let broadcast n vect = apply (put (apply (mkpar (fun pid v dst ->
   if pid=n then Some v else None)) vect)) (replicate n);;
val broadcast : int -> 'a par -> 'a option par
```

Now we give a trivial use of the `get` operation where the drawback of this operator is not significant.

Shift right Shift right values on a parallel vector modulo the number of processes is used in few BSP algorithms where each process has to deal with the data of its predecessor:



It could be done by a unique super-step by using an application of the `get` operator. We can prove the desired and certified expression for any parallel vectors:

$$\forall Data\ vec : (VectorData) \Rightarrow \exists res \text{ as } \forall i(0 \leq i < p)$$
$$(i = p-1 \rightarrow (res[i] = Some(vec[0]))) \vee (i \neq p-1 \rightarrow (res[i] = Some(vec[i-1])))$$

which could be proved by cases on i. The realizer is as follows:

```
let shift_right vect = get vect (mkpar (fun i ->
                            if i=0 then bsp_p()-1 else i-1))
val shift_right : 'a par -> 'a par
```

The `get` operator has proven to be sufficient to express all the BSP algorithms but not with an optimal cost. We give here an example, where the use of the `put` is essential to keep the efficiency of the algorithm.

Total exchange In a total exchange, each process communicates its own value to all the other processes. This could be expressed (using the **Coq**'s syntax) as:

```
Definition total_exchange : (Data:Set; v_data:(Vector Data))
{res:(Vector (Z->(Option Data))) | (i:Z) (H_i:‘0<=i<(bsp_p tt)‘)
(param:Z) if (within_bound param)
  then [H1] (((at (Z->(Option Data)) res i H_i) param)
                    = (Some Data (at Data v_data param H1)))
else [H2](((at (Z->(Option Data)) res i H_i) param)=(None Data))}.
```

The result of the specification of this function is a parallel vector containing functions. The application of one of these functions, for example the function f_i at process i, to j will give the value received by process i from process j. This specification could be proved by cases on `param`. Like in the expression of broadcast, the specification expresses that the result is `None` when `param` is not the "pid" of a process. The extracted program is:

```
let total_exchange v_data =
    put (apply (mkpar (fun a val pid -> Some val)) v_data);;
val total_exchange : 'a par -> (int -> 'a option) par
```

4.1 A certified `scan`

The `scan` function is also a classical parallel algorithm. It uses an operator R and (`scan` R $\langle x_0, \ldots, x_{p-1}\rangle$) evaluates to $\langle s_0, \ldots, s_{p-1}\rangle$ where $s_i = R_{k\leq i} x_k$. The specification of this function is:

$$\forall \text{Data}\ \forall vect\ \forall R : Data \to Data \to Data \Rightarrow$$
$$\exists res\ \text{as}\ \forall i\ H : (0 \leq i < p)\ (k\_first\_R\ T\ R\ res\ i\ H\ res[i])$$

using the following inductive type:

```
Inductive  k_first_R  [T:Set][R:T->T->T] [v:(Vector T)]
              : (k:Z) ‘0<=k<(bsp_p tt)‘ -> (res:T) Prop:=
   k_zero: (k_first_R T R v ‘0‘ H_0 (at T v ‘0‘ H_0))
 | k_rec : (k:Z) (H_r:‘0<=k<(bsp_p tt)‘)  (a:T) (H_sub:‘k>0‘)
                (k_first_R T R v ‘k-1‘ (p_sub k H_r H_sub)  a)
             -> (k_first_R T R v ‘k‘ H_r (R a (at T v ‘k‘ H_r))).
```

which gives the application of the operator R to the k first values of the parallel vector (`p_sub` is the a technical lemma which transforms a proof of $0 \leq k < p$ and $k > 0$ to $0 \leq k - 1 < p$). Because it is a logical inductive, it does not appear in the program at the extraction time. It is only used to verify formal properties. So we can use the "access" function without problem of nested of parallelism. There are differents proofs (with its computational part) of this specification: direct algorithm and one with a logarithm number of super-steps. The direct algorithm is naturally the most simple to prove. For the second, we used a "well-founded relation" [51] on the number which is growth from 0 to p. To end this overview of BSP expressions, we give an interesting result which has not been formally proved before: the possibility to express the `get` operation with the `put` one.

A new implementation of get At a communication level, there are mainly two approaches: *put* and *send*. As far as execution is concerned, it is better to put than to get. For a simple reason, get is implemented with two puts and involves two super-steps: a process sends a *request* to another designated process and then receives back a *reply* from it once the request is finally processed. Remark, the situation is just the opposite at the programming level (see the shift-right expression): each process controls its own communications (if it does not want to receive a new value, it just does not ask at the "get step"). The reply, which uses the result of the request to send its own value to a process which needs it. The implementation of the get operator is simply an application of the reply expression to the vector of process names:

```
let new_get_one datas srcs =
 let request = put(parfun (fun i dst->if dst=i then Some()
                                                 else None) srcs)
  and replace_by_data =
    parfun2 (fun f d dst->match(f dst) with
                              Some() -> Some d
                            | _      -> None)
in let reply = fun vect -> put(replace_by_data vect datas) in
  parfun (fun(Some x)->x) (apply (reply request) srcs)
val new_get : 'a par -> int par -> 'a option par
```

The request needs a super-step to ask to each process which of them send values or not. After, the reply expression sends the values and needs a second super-step (`parfun2` apply a function to the elements of two parallel vectors). The formal properties that need to be proved are the same as the axiom of the `get` operator. It could be proved by case and induction on the elements of the process names vector (`vect_n`). The proof obligations related to the indices are easily discharged by the useful arithmetic tactic of **Coq**, *Omega*, all the necessary inequalities being available from the context when the proof is required. This proof justifies that the `get` operator could be implemented as a combination of two `put` operations.

5 Related work

The first formal semantics of BSP was an axiomatic logic and parallel explicit semantics of the BSP model for a share-memory programming implementation [26]. It is based on mapping each process to a trace sequence which may be combined to determine composite behavior. Unfortunately, no programs have been certified with this set of algebraic laws and no implementation of the mathematical model of the specifications have been made. But the idea suggested the possibility to prove parallel algorithms with the BSP models while preserving the cost model. In [46], the presented semantics allows subset synchronization. This feature is not a part of the BSP model and there are many drawbacks to add it [22]. In another approach [48], the reasoning is made using a sequence of global

(parallel) state transformation. It eases the reasoning because whole parallel operations can be described as a global state transformation. This paper shows the refinement of a sequential version of the Floyd's shortest path algorithm to a BSP version. All those approaches are based on imperative languages.

Our approach has the following advantages. It is based on a functional language so it eases the reasoning. Using the Coq proof assistant our proofs are partially automated which is not the case in other approaches. Moreover we can generate programs using Coq. In our framework the compiler and runtime system are partially certified. We also believe that the use of more complex data structures will be rather easy in the case of BSML, but would be very complex in other approaches.

6 Conclusions and Future Work

We have formalized the parallel operations of the BSML language. Using this formalization we proved properties of very often used BSML programs. This allows to validate those programs and even to find mistakes in the hand-written proofs.

Several directions can be followed for future work : validation of more complex BSP programs as parallel sorts [50, 49, 52], our goal is to have a certified `BSMLlib` library (including its standard library) ; extension of this framework to include imperative features (like [26]) using work done on sequential programs [14] and have a software for certification of BSML programs (like in [15]) ; studying of the possibility to prove cost formulas.

We also continue our work on the certification of the environment. In particular, the parallel abstract machine proved with respect to the BSλ-calculus is a SECD based parallel machine, which is of course not the one used in the implementation of the `BSMLlib`. For this reason, we designed a parallel abstract machine based on the Zinc Abstract Machine [18] used in the current implementation of Objective Caml [29]. This machine has to be proved correct. We may use a version of the BSλ-calculus with explicit substitution in order to follow the methodology presented in [25] for the validation of several abstract machines with respect to a calculus of explicit substitution.

Future work will also consider extensions of the BSML framework with parallel compositions: the parallel juxtaposition [33] which allows to divide the network in subnetworks while preserving the BSP cost model and the parallel superposition [35] which allows to have in a pure functional setting a constructor which can be used to run to "BSP threads". This new construction is particularly interesting for multiprogramming and thus is a first step towards the use of BSML for Grid computing.

Acknowledgments This work is supported by the ACI Grid program from the French Ministry of Research, under the project CARAML (www.caraml.org).

References

1. G. Akerholt, K. Hammond, S. Peyton-Jones, and P. Trinder. Processing transactions on GRIP, a parallel graph reducer. In Bode et al. [6].
2. Arvind and R. Nikhil. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4), 1989.
3. O. Ballereau, F. Loulergue, and G. Hains. High-level BSP Programming: BSML and BSλ. In G. Michaelson and Ph. Trinder, editors, *Trends in Functional Programming*, pages 29–38. Intellect Books, 2000.
4. M. Bamha, F. Bentayeb, and G. Hains. An efficient scalable parallel view maintenance algorithm for shared nothing multi-processor machines. In Trevor Bench-Capon, Giovanni Soda, and A Min Tjoa, editors, *10th International Conference on Database and Expert Systems Applications, DEXA'99*, number 1677 in Lecture Notes in Computer Science, pages 616–625, Florence, Italy, August 30 – September 3 1999. Springer-Verlag.
5. R. H. Bisseling and W. F. McColl. Scientific computing on bulk synchronous parallel architectures. In B. Pehrson and I. Simon, editors, *Technology and Foundations: Information Processing '94, Vol. I*, volume 51 of *IFIP Transactions A*, pages 509–514. Elsevier Science Publishers, Amsterdam, 1994.
6. A. Bode, M. Reeve, and G. Wolf, editors. *PARLE'93, Parallel Architectures and Languages Europe*, number 694 in Lecture Notes in Computer Science, Munich, June 1993. Springer.
7. G.-H. Botorog and H. Kuchen. Efficient high-level parallel programming. *Theoretical Computer Science*, 196:71–107, 1998.
8. A. Braud and C. Vrain. A parallel genetic algorithm based on the BSP model. In *Evolutionary Computation and Parallel Processing GECCO & AAAI Workshop*, Orlando (Florida), USA, 1999. To appear.
9. M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.
10. Th. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 37(2-3), 1988.
11. J. Darlington, A. J. Field, P. G. Harrison, P. Kelly, D. Sharp, Q. Wu, and R. While. Parallel programming using skeleton functions. In Bode et al. [6].
12. D. C. Dracopoulos and S. Kent. Speeding up genetic programming: A parallel BSP implementation. In *First Annual Conference on Genetic Programming*. MIT Press, July 1996.
13. F. Dehne (Guest Editor). Special issue on coarse-grained parallel algorithms. *Algorithmica*, 14:173–421, 1999.
14. J.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 2001. To appear.
15. J.-C. Filliâtre. *Why: a software verification tool*. http://why.lri.fr, 2003.
16. C. Foisy and E. Chailloux. Caml Flight: a portable SPMD extension of ML for distributed memory multiprocessors. In A. W. Böhm and J. T. Feo, editors, *Workshop on High Performance Functionnal Computing*, Denver, Colorado, April 1995. Lawrence Livermore National Laboratory, USA.
17. F. Gava. Formal Proofs of BSML Programs in Coq. Web pages at www.univ-paris12.fr/lacl/gava, 2003.
18. F. Gava and F. Loulergue. A Parallel Virtual Machine for Bulk Synchronous Parallel ML. In *International Conference on Computational Science (ICCS 2003)*. Springer Verlag, june 2003. to appear.

19. F. Gava and F. Loulergue. A Polymorphic Type System for Bulk Synchronous Parallel ML. In *Seventh International Conference on Parallel Computing Technologies (PaCT 2003)*, LNCS. Springer Verlag, 2003. to appear.
20. A. V. Gerbessiotis and L. G. Valiant. Direct Bulk-Synchronous Parallel Algorithms. *Journal of Parallel and Distributed Computing*, 22:251–267, 1994.
21. L. Granvilliers, G. Hains, Q. Miller, and N. Romero. A system for the high-level parallelization and cooperation of constraint solvers. In Y. Pan, S. G. Akl, and K. Li, editors, *Proceedings of International Conference on Parallel and Distributed Computing and Systems (PDCS)*, pages 596–601, Las Vegas, USA, 1998. IASTED/ACTA Press.
22. G. Hains. Subset synchronization in BSP computing. In H.R.Arabnia, editor, *PDPTA'98 International Conference on Parallel and Distributed Processing Techniques and Applications*, volume I, pages 242–246, Las Vegas, July 1998. CSREA Press.
23. G. Hains and C. Foisy. The Data-Parallel Categorical Abstract Machine. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE'93*, number 694 in LNCS, pages 56–67. Springer, 1993.
24. G. Hains and F. Loulergue. Functional Bulk Synchronous Parallel Programming using the BSMLlib Library. In S. Gorlatch and C. Lengauer, editors, *Constructive Methods for Parallel Programming*, Advances in Computation: Theory and Practice, pages 165–178. Nova Science Publishers, august 2002.
25. T. Hardin, L. Maranget, and L. Pagano. Functional runtime systems within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–176, 1998.
26. Lei Chen He Jifeng, Quentin Miller. Algebraic Laws for BSP Programming. In L. Bougé and Y. Robert, editors, *Euro-Par'96 Parallel Processing*, number 1124 in Lecture Notes in Computer Science, pages 359–368, Lyon, France, August 1996. Springer.
27. J.M.D. Hill, W.F. McColl, and al. BSPlib: The BSP Programming Library. *Parallel Computing*, 24:1947–1980, 1998.
28. Guy Horvitz and Rob H. Bisseling. Designing a BSP version of ScaLAPACK. In Bruce Hendrickson et al., editor, *Proceedings Ninth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, Philadelphia, PA, 1999.
29. X. Leroy. The ZINC experiment: An economical implementation of the ML language. Technical Report 117, INRIA, 1991.
30. Xavier Leroy. The Objective Caml System 3.06, 2002. web pages at www.ocaml.org.
31. F. Loulergue. BSλ_p: Functional BSP Programs on Enumerated Vectors. In J. Kazuki, editor, *International Symposium on High Performance Computing*, number 1940 in Lecture Notes in Computer Science, pages 355–363. Springer, October 2000.
32. F. Loulergue. Distributed Evaluation of Functional BSP Programs. *Parallel Processing Letters*, (4):423–437, 2001.
33. F. Loulergue. Parallel Composition and Bulk Synchronous Parallel Functional Programming. In S. Gilmore, editor, *Trends in Functional Programming, Volume 2*, pages 77–88. Intellect Books, 2001.
34. F. Loulergue. Implementation of a Functional Bulk Synchronous Parallel Programming Library. In 14^{th} *IASTED International Conference on Parallel and Distributed Computing Systems*, pages 452–457. ACTA Press, 2002.
35. F. Loulergue. Parallel Superposition for Bulk Synchronous Parallel ML. In *International Conference on Computational Science (ICCS 2003)*. Springer Verlag, june 2003. to appear.

36. F. Loulergue, G. Hains, and C. Foisy. A Calculus of Functional BSP Programs. *Science of Computer Programming*, 37(1-3):253–277, 2000.
37. W. F. McColl. Scalability, portability and predictability: The BSP approach to parallel programming. *Future Generation Computer Systems*, 12:265–272, 1996.
38. W. F. McColl. Universal computing. In L. Bouge and al., editors, *Proc. Euro-Par '96*, volume 1123 of *LNCS*, pages 25–36. Springer-Verlag, 1996.
39. A. Merlin, G. Hains, and F. Loulergue. A SPMD Environment Machine for Functional BSP Programs. In *Proceedings of the Third Scottish Functional Programming Workshop*, august 2001.
40. P. Panangaden and J. Reppy. The essence of concurrent ML. In F. Nielson, editor, *ML with Concurrency*, Monographs in Computer Science. Springer, 1996.
41. C. Parent. Developing certified programs in the system coq: The program tactic. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs*, pages 291–312. Springer, Berlin, Heidelberg, 1993.
42. C. Paulin-Mohring. Extracting $\mathcal{F}_w$'s programs from proofs in the calculus of constructions. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, January 1989.
43. S. Pelagatti. *Structured Development of Parallel Programs.* Taylor & Francis, 1998.
44. R. O. Rogers and D. B. Skillicorn. Using then BSP cost model to optimise parallel neural network training. *Futur Generation Computer Systems*, 14(5-6):409–424, 1998.
45. D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3), 1997.
46. D.B. Skillicorn. Building BSP Progams Using the Refinement Calculus . In *Formal Methods for Parallel Programming and Applications workshop at IPPS/SPDP'98*, 1998.
47. M. Snir and W. Gropp. *MPI the Complete Reference.* MIT Press, 1998.
48. A. Stewart, M. Clint, and J. Gabarró. Algebraic Rules for Reasoning about BSP Programs. In S. Gorlatch and C. Lengauer, editors, *Constructive Methods for Parallel Programmin*, Advances in Computation: Theory and Practice. Nova Science Publishers, august 2002.
49. K. R. Sujithan and J. M. D. Hill. Collection types for database programming in the BSP model. In *Fifth Euromicro Workshop on Parallel and Distributed Processing.* IEEE CS Press, January 1997.
50. K. Ronald Sujithan. Towards a scalable, parallel object database - the bulk synchronous parallel approach,. Technical Report PRG-TR-17-96, Programming Research Group, Oxford University Computing Laboratory, August 1996.
51. The Coq's team. *The Coq Proof Assistant.* http://coq.inria.fr, 2003.
52. A. Tiskin. *The Design and Analysis of Bulk-Synchronous Parallel Algorithms.* PhD thesis, Oxford University Computing Laboratory, 1998.
53. Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.

The Semantics of C++ Data Types: Towards Verifying low-level System Components

Michael Hohmuth Hendrik Tews

Dresden University of Technology
Department of Computer Science

vfiasco@os.inf.tu-dresden.de

May 30, 2003

In order to formally reason about low-level system programs one needs a semantics (for the programming language in question) that can deal with programs that are *not statically type-correct.* For system-level programs, the semantics must deal with such heretical constructs like casting integers to pointers and converting pointers between incompatible base types.

In this paper we describe a formal semantics for the data types of the C++ programming language that is suitable for low-level programs in the above sense. This work is part of a semantics for a large subset of the C++ programming language developed in the VFiasco project. In the VFiasco project we aim at the verification of substantial properties of the Fiasco microkernel, which is written in C++.

1 Introduction

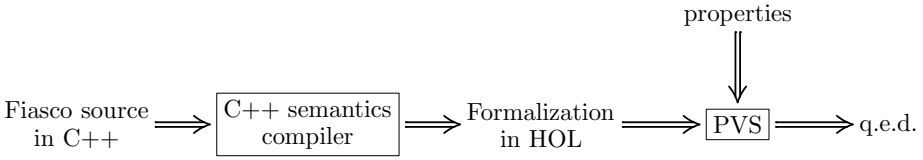
The VFiasco [21] project aims at the verification of substantial properties of the Fiasco [9] microkernel for x86 PC hardware (more precisely for IA32-based systems). Fiasco is a real-time microkernel operating system. It has been developed in the context of the DROPS project [7] and supports the flexible construction of applications with security or quality of service requirements. As a microkernel, Fiasco has a minimal interface and supports only the absolutely necessary operating-system functionality. There are, for instance, no device drivers included in Fiasco. For legacy applications it is possible to boot Linux on top of Fiasco [8].

Fiasco is almost entirely written in C++ [20]. Only those operations that cannot be performed in C++ (like accessing CPU control registers) are written in inline assembler. The properties that we attempt the prove in the VFiasco project include the following:

This work was supported by the Deutsche Forschungsgemeinschaft (DFG) through DFG grant Re 874/2-1.

- Fiasco's internal page-fault handler terminates for all kernel page faults
- Fiasco's internal memory allocation works correctly

For the verification, we plan to employ a state-of-the-art general-purpose theorem prover. Our current development focuses on PVS [17]. For the verification, the C++ source code of Fiasco will be fed into a semantics compiler that generates the semantics of the input as shallow embedding[1] in the higher-order logic of PVS. Specifications and properties will be formulated directly in the logic of PVS. The whole verification process can be depicted as follows:



The semantics compiler is currently developed on the basis of the OpenC++ package [4]. For the translation of Fiasco's source code we need a semantics for a substantial part of C++.

In this paper we concentrate on the built-in types and the type constructions (like structs and unions) of C++. Other details of our semantics of C++ will be described elsewhere, see [10, 11]. We base our formalisation of the data types on the C++ ISO Standard [5]. In the following we call this document simply *the standard*. The notation §$n.m(k)$ refers to Section $n.m$, paragraph k in the standard. In the standard the data types are mainly described in §3.9 and §9. We have the following requirements for our semantics:

1. The semantics is suitable for reasoning about low-level systems programs, especially operating systems written in C++, like Fiasco.
2. The semantics scales well to the formalisation of a specific C++ implementation that determines the behaviour of certain language constructs that is unspecified in the standard.
3. The semantics supports popular unsafe C++ programming idioms (like allocating a `char` array and casting its base address into a specific pointer type) together with programming idioms that are necessary for operating-systems programming (like casting unsigned integers to pointers).
4. The semantics is sensible enough to detect subtle programming errors, like reading data from uninitialised memory.

The motivation for these requirements is clear from the context of this work: After all we want to use the semantics in a concrete verification project. Let us now discuss the implications of these requirements.

The first consequence of Requirement 1 (reasoning about low level code) is that we cannot use the traditional way of modelling dynamically allocated structures. For instance in [3, 15]

[1] In a shallow embedding an external tool translates the sources into their semantics. In contrast, in a deep embedding one can represent phrases of the source in the logic and the semantic function is expressed inside the logic.

dynamically allocated structures are modelled as functions of the form *heap* : *address* $\longrightarrow$ *value-type*. In this approach the heap is implicitly typed, making it impossible to describe programs which possibly contain type errors (like reading an integer from a memory location that contains a string). However, one of the goals of the VFiasco project is to prove the absence of such typing errors. This is impossible in an implicitly-typed semantics. Our model of the heap must support operations like pointer-type conversion and also interpreting the same data in different types.

For the second consequence of the first requirement recall again our long term goal: Verifying Fiasco. For that we are currently developing an x86 hardware model in PVS. The hardware model gives an abstraction of the execution environment of x86 compatible processors. A state of the hardware model will be an abstraction of the state of real-world PC. Such a state contains a snapshot of the physical memory together with some important control registers on the CPU (like `cr3`, which determines the base address of the current page directory for the virtual-memory address translation). For the verification, the semantics of the Fiasco sources will be interpreted on the hardware model. In order to ensure that the verification results apply to the real-world Fiasco, it is necessary that our hardware model models x86 hardware — and nothing more. The conclusion is that the semantics of data types must not add information to the states of the hardware model, like, for instance, associating type tags to memory locations.

The second requirement (scalability of the semantics) originates from the following: The C++ standard is very vague about almost every C++ construct. For the built-in integer types it does not specify which values they can represent, for signed integer types it is left open if they can represent negative numbers. One of the few things that are required is that all built-in integer types are at least 7 bits wide.[2] A specific C++ implementation can define things that are left open in the standard. For instance the gcc compiler uses 8 bits for `char` and 32 bits for `int`. It is impossible to write (and verify!) an operating system without such implementation-specific knowledge. For instance, Fiasco relies on the fact that the standard conversion from any pointer type to `unsigned` and back is the identity function on the underlying bit representation. The data-type semantics in this paper will easily scale to (possibly different) implementations. The base version of the semantics formalises the C++ standard. From the base version one can derive an implementation-specific version by just adding the additional properties.

Requirement 3 (support for unsafe idioms) is (again) motivated by our long-term goal in the VFiasco project. The semantics must support all the program idioms that are necessary for writing an operating system. This includes casting (seemingly) arbitrary unsigned integers into pointers and dereferencing these pointers.

Last but not least we want the semantics to catch all possible programming errors (Requirement 4. Consider the following C++ statement[3]

```
T a = * reinterpret_cast<T *>(50)
```

It accesses an object of type `T` that is located at address 50. At the time where this statement is executed there might or might not be a valid object of type `T` stored at address

[2] Because they all can represent the 96 characters from the *basic source character set*, see §§ 2.2.1, 3.9.1.2

[3] The C++ standard says that the behaviour of this statement is undefined. However for gcc it is well defined.

50. Statements of this kind are necessary in an operating system, for instance in the code that traverses page directories. Therefore we need a semantics for data types that permits the above statement (with the expected semantics) provided one can prove that there is a valid object of type `T` stored at address 50. However, we also want to catch programming errors where the above statement is executed before the memory at address 50 is properly initialised. Therefore, without any knowledge about the memory contents one should not be able to derive anything about the above statement (not even that it does not crash the hardware).

For a second example about which programming errors we want to catch, consider the following piece of code:

```
unsigned a[2] = { 1, 2 };
unsigned b[3];
memcpy(static_cast<char *>(static_cast<void *>(b)) + 1, a, sizeof(a));
unsigned c = * static_cast<int *> (static_cast<void *>
                        (static_cast<char *> (static_cast<void *> (b)) + 1));
unsigned d = b[1];
```

The `memcpy` function copies the two integers 1 and 2 in array `a` into array `b`. Note that the copy is displaced by an offset of 1. According to the C++ standard such a `memcpy` is legitimate. Moreover, it specifies that the fourth statement (which reads the integer 1 at its new address) initialises `c` correctly with 1 (§3.9.3) provided the alignment requirements are met. For any compiler we know, also the last statement will execute without problems. On x86 hardware (with little endian byte ordering) it will combine the most significant byte of 1 (which is 0) together with the three less significant bytes of 2 to a result of 512 (decimal). However, the C++ standard leaves the behaviour of the last statement open (it could potentially crash the machine). We would like to view the evaluation of `b[1]` as an error (because it reads an integer where none has been stored before). In the semantics we present here, one cannot even prove that the last program does not crash.

Related Work Traditionally, program verification focuses on well-typed programming languages (see for instance [2]), which have a relatively simple semantics. However, as explained before, in the VFiasco project we *need* to reason about an unsafe programming language. Recently, in the work on proof-carrying code, type systems and semantics have been developed for assembly languages [16]. However, the type systems developed for proof-carrying code are not well suited for complex analysis. With our present work we try to find a compromise between reasoning on an abstract level and the ability to treat possibly illtyped programs.

A semantics for C and C++ has also been described in the framework of abstract state machines [6, 23]. There exist simulators for abstract state machines, but, to our knowledge, no theorem-proving support.

Our work is very much inspired by the work in the LOOP project for reasoning about Java programs [22]. Jacobs studies in [14] the integral types of Java. The main difference to our work is (apart from the source languages) that for Java's integral types *nothing* is left unspecified. Therefore Jacobs can model his semantics as a definitional extention in PVS. However, with Jacobs semantics one cannot reason about the absence of over- or underflows.

Acknowledgements We would like to thank Sarah Hoffmann, Matthias Daum, and Shane G. Stephens for many discussions on the subject.

2 General Properties of C++ Data Types

In this section we analyse the general properties of data types in C++ and explain the general approach of our semantics. In the standard one can distinguish three types of descriptions of the behaviour of an operation:

1. The standard leaves the behaviour (explicitly or implicitly) unspecified (permitting the compiler to reject the program, the program to crash, or to continue with the most sensible result).
2. The standard says that the evaluation terminates normally but the results are unspecified (the program must not crash but one cannot rely on the result).
3. The standard describes the behaviour in detail.

For the first two points the standard distinguishes between *implementation defined* and unspecified behaviour. This distinction is not important for us.

For the development of the semantics we use the following approach: For a description of Type 1 or 2 we use an undetermined axiomatic specification with uninterpreted constants. For Type 2 we add appropriate additional axioms that ensure termination. For type 3 we use a definition. The effect is as follows: The available proof power correlates precisely to what one knows about an execution of the program on an arbitrarily chosen C++ implementation (including all hypothetical ones). Consider, for instance, a program that uses an unsigned-integer operation that is only guaranteed to terminate. The result of that operation will subsequently be divided by 2 and stored in some variable v. For such a program one can prove termination under the precondition that the unspecified integer operation does not return zero. Further, one can derive that the value in v is less than or equal to maximal unsigned integer divided by 2. Any attempt to prove something more specific will fail.

The standard divides all types into POD (*plain old data*) types and non-POD types (see §§3.9 (10), 9 (4)). Basically, POD types are those types that are compatible with C. That is, all arithmetic types, structs, and unions are POD, provided they do not contain any virtual functions, constructors, destructors and the like.

The standard describes the C++ memory and object model in §1.7 and §1.8. The interesting point for us is that objects (i.e., memory representations of values of C++ types) have an address, a type, and occupy some memory. Objects of POD types are stored contiguously in memory and they can be copied or moved (§3.9 (2)). Objects of non-POD type cannot be moved and one has to use the copy constructor or assignment operator to copy them. Some common aspects of all types are described in §3.9, the built-in types (called *fundamental types*) are described in §3.9.1. The properties of classes, structures, unions, and bit fields are laid out in §9.

For any type T the standard distinguishes between its *value type* (i.e., the set of values of T), the *value representation* (i.e., the bit string that represents a value), and the *object representation* (i.e., the bit string that is stored in memory). There are special requirements for the three character types that we discuss below. For all other types the object representation can be different from the value representation (which is indeed the case on

all little-endian machines). The mapping from the value representation to the values might not be injective (for instance it is possible to use one-complement encoding for integers with two different representations of zero: $+0$ and -0).

The value representation is only used in the description of the bit-wise operations in §§5.8, 5.11–5.13. However, the standard either defines the bitwise operations precisely in terms of the argument values or leaves the result undefined. We therefore decided to ignore the value representation. In our semantics we model values and their object representation.

To relate object and value representation the standard states in §3.9 (4):

> *For POD types, the value representation is a set of bits in the object representation that determines a value ...*

It is a fundamental observation that the preceding citation it *the only* requirement that the standard puts on the connection of object and value representation. In particular, the object representation might contain more bits than the value representation. These additional bits can be used for arbitrary purposes. The C++ runtime system can, for instance, use them to encode the object type into the object representation and to perform Lisp-like runtime type checking. For non-POD types one could even encode the memory location of the object into the object representation to detect (at runtime) if the object has been illegally moved with `memcpy`. We can therefore derive the following observation.

Observation 1 *A C++ program that does not crash on any C++ implementation must be type correct in the following sense:*

- *For any type T which is not a character type, it accesses objects of type T only at memory locations that contain a correctly initialised object of type T.*
- *It accesses members of any non-POD type T only at memory locations that have been initialised by a constructor or assignment operator for T.*

To summarise: For the semantics of a data type we need a value type and functions that translate values into their object representation and back. These functions (and if necessary the value type) must be undetermined to a certain extent to model all possible C++ implementations. To prove type correctness of a C++ program it is then sufficient to prove that the program does not crash.

3 Common PVS Formalisation

This section describes the common infrastructure that we use for all types. Let us fix some notions before we turn to the PVS sources. Consider a C++ data type T. Any C++ implementation defines a value type and an object representation for T. We call such an implementation of a type T a *model of* T. As semantics of T we develop a specification for all possible models of T.

Our semantics will be independent of the underlying (model of the) memory. This makes it possible to reuse the same data-type semantics for all the memory abstractions that we are developing in the VFiasco project [12]. However, it may be helpful to imagine an

oversimplified memory model, consisting of a type State containing all memory states, and two functions (in PVS syntax):

```
memory_read : [State, Address -> Byte]
memory_write : [State, Address, Byte -> State]
```

Here, Byte is the type of all values that can be stored in a byte and Address is the type of addresses (as a subtype of natural numbers). The precise definition of these two types does not matter for the contents of this paper.

These memory-access functions can be extended in the obvious way to

```
memory_read_list : [State, Address, nat -> list[Byte]]
memory_write_list : [State, Address, list[Byte] -> State]
```

In our semantics, every C++ data type T is modelled in PVS by its value type together with a record that contains the basic common operations. The operations in the record depend on the value type. Its definition in PVS is:

```
Data_type_structure : Type = [#
  size : nat,
  to_byte : [Data, Address -> list[Byte]],
  from_byte : [list[Byte], Address -> lift[Data]]      #]
```

The types Address and Byte are as before. The type parameter Data stands for the value type of T. The first field size contains the length of the object representation (in bytes). It corresponds to the `sizeof` operator of C++. The functions to_byte and from_byte convert values into their object representation and back. As an additional argument they take the address of the object representation in the memory. This way the function to_byte could encode the address in the object representation and from_byte can check it to prevent illegal copying. Naturally from_byte is a partial function that is only defined on valid object representations. We use the traditional PVS approach and represent a partial function $X \rightharpoonup Y$ as $X \to \mathsf{lift}[Y]$ in PVS.[4]

For any model of a C++ type we require some basic properties. They are combined in a predicate as follows:

```
data_type? : PRED[Data_type_structure] = Lambda(ct : Data_type_structure) :
  (Forall(d : Data, a : Address) : length(to_byte(ct)(d,a)) = size(ct)) And
  (Forall(d : Data, a : Address) : up?(from_byte(ct)(to_byte(ct)(d,a), a))) And
  (Forall(d : Data, a : Address) : down(from_byte(ct)(to_byte(ct)(d,a), a)) = d ) And
  (Forall(l : list[Byte], a : Address) : up?(from_byte(ct)(l, a)) Implies length(l) = size(ct))
```

This predicate requires that

- the object representation of all values is size(ct) bytes long.
- from_byte is a left inverse of to_byte (i.e., from_byte is defined on all results of to_byte and yields the original value).
- from_byte fails on objects of the wrong size

[4] The type constructor lift corresponds to option in Isabelle. In PVS (the representation of) a partial function returns bottom if it is undefined and up(-) otherwise.

In contrast to the standard that requires all objects to occupy some memory (§1.8 (5)) we do not require the size field to be positive. A size of zero makes sense for `void`, see Subsection 4.5. Inhabitants of the predicate subtype (data_type?[Data]) are models of C++ types with Data as value type.

For objects of a POD-type the standard requires that one can copy the object to a different memory location. Therefore we strengthen the data_type? predicate for POD types:

```
pod_data_type? : PRED[Data_type_structure] =
  Lambda(ct : Data_type_structure) : data_type?(ct) And
    (Forall(d : Data, a1,a2 : Address) : up?(from_byte(ct)(to_byte(ct)(d,a1), a2)))
```

This addition ensures that the from_byte function is successful regardless of the address on which we find a valid object representation.[5]

For any model dt of type (data_type?[Data]) one can now define two functions that attempt to read or write a value into the memory:

```
read_data(dt)(s : State, addr : Address) : lift[Data] =
  from_byte(dt)(memory_read_list(s, size(dt), addr), addr)

write_data(dt)(s : State, addr : Address, data : Data) : State =
  memory_write_list(s, addr, to_byte(dt)(data, addr))
```

We can now precisely define what we mean with the semantics and a model of a data type. Note that in PVS any constant or function definition[6] determines a specific value, even if the axiom of choice is used in the definition. Therefore the only way to achieve the needed undeterminedness in our semantics is to use declarations in an axiomatic specification. This approach requires special efforts to maintain soundness.

Definition 2 (Semantics) The semantics of a C++ type T in PVS consists of

- a type definition Semantics_T for the value type
- a constant declaration of type Data_type_structure providing the common operations; for non-POD types this constant must be in data_type?[Semantics_T] and for POD types in pod_data_type?[Semantics_T]
- a finite number of axioms stating additional properties

Note that one can add further axioms to any existing semantics of a type T to obtain a new semantics of T. This feature ensures scalability in the sense of Requirement 2 from the introduction. We will exploit this feature in the following way: First we develop a semantics for the data types as described in the standard. In a second step we derive a specific semantics for the GNU gcc compiler by adding more axioms.

For some types the standard leaves the value type implementation-defined. A semantics of such a type will fix the the super type of all value types of all models. The value type Semantics_T can then be defined as a predicate subtype that involves an uninterpreted constant (such that the precise range of Semantics_T stays undefined). We use this technique for all signed integer types, see Subsection 4.2 below.

[5] We ignore alignment issues here, see Subsection 6.

[6] Here we mean *interpreted constant definitions* in the sense of the PVS Language Reference [19]. We refer to *Uninterpreted constant definitions* as *declarations*.

Definition 3 (Model) A model of a C++ type T in PVS consists of a type definition Model_T and a constant definition of type (data_type?[Model_T]), or, if T is POD, (pod_data_type?[Model_T]) such that these two definitions

- do not involve uninterpreted constants or types
- fulfil the axioms from the semantics of T

In principle the model relationship can be established with the notion of theory interpretations [18] that have been introduced in PVS recently. A theory interpretation provides definitions for uninterpreted constants. Axioms involving the constants reappear as proof obligations. With theory interpretations it is possible to establish the soundness of an axiomatic specification *whithin* PVS. However, in the current PVS version there remain a few issues[7] to be resolved before theory interpretations can be applied in our context.

To maintain consistency we develop a model for every data type together with its semantics. We also plan that our C++ semantics compiler will generate *both* the semantics and a model for every used data type.

Closely related with the data types are the C++ standard conversions. Here we treat the semantics of data types and that of the conversions separately although they are very closely related: Sometimes the standard specifies a property of a data type in an indirect way by putting requirements on the conversion functions.

The semantics of a conversion function is an appropriately typed constant declaration. It is accompanied with axioms in case the standard restricts the behaviour of the conversion. As one expects, a model of a conversion is a function definition.

4 Fundamental Types

In this section we describe the semantics of the fundamental types of C++.

4.1 Booleans

For booleans the standard is very clear: "Values of type bool are either true or false." (§3.9.1 (6)). For the semantics of `bool` we set

```
Semantics_bool: Type = bool
dt_bool_exists : Axiom Exists (x: (pod_data_type?[Semantics_bool])): true
dt_bool : (pod_data_type?[Semantics_bool])
```

We need the axiom on the second line to discharge the existence TCC[8] for the declaration dt_bool. Note that the semantics does not stipulate that the value **true** is represented as a non-zero byte. The standard only says that the result of *converting* **true** into an integer type is one.

[7] See PVS bug reports 757–760 on `http://pvs.csl.sri.com/cgi-bin/pvs/pvs-bug-list/`.

[8] A *Type check condition* (TCC) is a proof obligation generated by the type checker when it cannot decide whether an expression is type-correct or not. For PVS TCC's are necessary because the type system is not decidable.

For a model of `bool` we only have to define dt_bool (in a different theory):

```
dt_bool : (pod_data_type?[Semantics_bool]) = (#
  size := 1,
  to_byte := Lambda(b : bool, a : Address) : IF b Then (: 1 :) Else (: 0 :) Endif,
  from_byte := Lambda(bl : list[Byte], a : Address) :
                 IF bl = (: 1 :) Then up(true)
                 Elsif bl = (: 0 :) Then up(false)
                 Else bottom   Endif    #)
```

Note that for this definition PVS generates a TCC that requires us to prove that the POD data-type properties hold. Now one can use theory interpretations to show that the semantics of booleans is sound. A more simple-minded approach is to copy the only axiom from the semantics of booleans and prove it as a lemma:

```
dt_bool_exists : Challenge Exists (x: (data_type?[Semantics_bool])): true
```

4.2 Signed Integers

There are four signed integer types in C++: `signed char`, `short int`, `int`, and `long int`. Character types are treated separately in Subsection 4.4 because of their special requirements. For the other signed integer types the standard does not say much. For instance it is not required that they can hold negative numbers. We only show the semantics of `int` here, the semantics of the other types is very similar.

```
Cxx_Int : Theory
Begin
  int_bits: posnat
  semantics_int_pred : PRED[int]

  Importing Cxx_Sshort
  int_longer : Axiom sshort_bits <= int_bits
  int_contains_sshort : Axiom subset?(semantics_sshort_pred, semantics_int_pred)

  Semantics_int : Type = (semantics_int_pred)
  Importing Abstract_Data[Semantics_int]
  dt_int_exists : Axiom Exists (x: (pod_data_type?[Semantics_int])): True
  dt_int : (pod_data_type?[Semantics_int])
End Cxx_Int
```

The identifiers with sshort refer to the corresponding items from the semantics of `signed short`. First we declare the size of the value representation, this becomes important for the unsigned integer types, see below. We define the value type Semantics_int as a predicate subtype of the PVS integer type int. The axioms int_longer and int_contains_sshort formalise the requirement that "[short int] provides at least as much storage as [int]" (§3.9.1 (2)).

The standard further requires that the value representation uses a *pure binary numeration system* (§3.9.1 (7)). However, it is unclear to us in which way a program could rely on the use of a pure binary numeration system. Programs that do use integers usually rely on the fact that at least a certain interval can be represented in the integer types. Most C++ implementations specify the value type of the integer types. Therefore we do not bother to

axiomatise pure binary numeration systems. Instead we rely on additional assumptions on the value type of the integers. To obtain the integer type of the GNU C++ compiler on x86 one could use the following theory:

```
Gnu_IA32_Int : Theory
Begin
  Importing Cxx_Int
  int_bits_ia32 : Axiom int_bits = 32
  int_pred_ia32 : Axiom semantics_int_pred = { i : int | -2^31 <= i And i < 2^31}
End Gnu_IA32_Int
```

The models for the integer types are the obvious ones. We skip their presentation here.

4.3 Unsigned Integers

For each signed integer there is a corresponding unsigned integer. For the unsigned integer types the standard specifies arithmetic modulo 2^n, where n is the number of bits in the value representation. The formalisation of the type `unsigned` is as follows:

```
Cxx_Unsigned : Theory
Begin
  unsigned_bits: posnat
  semantics_unsigned_pred : PRED[nat] = { n : nat | n < 2^unsigned_bits }

  Importing Cxx_Int, Cxx_Ushort
  unsigned_same_size : Axiom unsigned_bits = int_bits
  unsigned_inclusion : Axiom
    Forall(i : Semantics_int) : i >= 0 Implies semantics_unsigned_pred(i)
  unsigned_longer : Lemma ushort_bits <= unsigned_bits
  unsigned_contains_ushort : Lemma
        subset?(semantics_ushort_pred, semantics_unsigned_pred)

  Semantics_unsigned : Type = (semantics_unsigned_pred)
  Importing Abstract_Data[Semantics_unsigned]
  dt_unsigned_exists : Axiom Exists (x: (pod_data_type?[Semantics_unsigned])): True
  dt_unsigned : (pod_data_type?[Semantics_unsigned])
End Cxx_Unsigned
```

For unsigned integers the value type depends only on the number of bits in the value representation. The two axioms unsigned_same_size and unsigned_inclusion formalise that the value representation of `int` and `unsigned int` has the same size and that a positive value of `int` is also a value of `unsigned int`. Other requirements of the standard follow now as lemma: for instance that the value representation of `unsigned int` is longer than that of `unsigned short`.

4.4 Character Types

There are three different character types in C++, `unsigned char`, `signed char`, and `char`. The value type of `char` coincides with either `signed char` or `unsigned char`. Note that character types are integer types, that is, their values are integers and not characters. The special property of the character types is that one can copy every POD object into an array

of a character type of sufficient length (see §3.9 (2) and issue 350 in [1]). This must work even if the memory of the source object has not been correctly initialised. The semantics of signed characters is as follows:

```
Cxx_Schar : Theory
Begin
  schar_bits: posnat
  semantics_schar_pred : PRED[int]
  Semantics_schar : Type = (semantics_schar_pred)                                  5
  Importing Abstract_Data[Semantics_schar]
  dt_schar_exists : Axiom Exists (x: (pod_data_type?[Semantics_schar])): True
  dt_schar : (pod_data_type?[Semantics_schar])

  dt_schar_from_byte_total: Axiom Forall (l: list[Byte], a : Address):
    length(l) = size(dt_schar) Implies up?(from_byte(dt_schar)(l, a))            10

  dt_schar_injective: Axiom Forall (l: list[Byte], a : Address):
    length(l) = size(dt_schar) Implies
      to_byte(dt_schar)(down(from_byte(dt_schar)(l, a)), a) = l
End Cxx_Schar
```

Up to line eight we have the usual semantics for signed integer types. The remainder formalises the ability to copy POD objects. The axiom on line nine expresses that one can interpret every piece of memory as signed character (the from_byte function never fails). The last axiom ensures that the data does not change when it is copied (the from_byte function is injective on byte lists of the right length).

Both properties need not hold for any non-character type T: There might be a bit pattern that does not describe a value of T, such that from_byte fails. There also might be two bit patterns that describe the same value (like $+0$ and -0 in a one's-complement representation), such that from_byte is not injective.

Our axiomatisation of signed characters implies that there is a bijective correspondence between the value type (Semantics_schar) and the object representation (byte lists of length size(dt_schar)). This can be proved within PVS (where List_len(l) is the type of lists of length l):

```
schar_iso : Lemma
  Exists(f : [Semantics_schar -> List_len[Byte](size(dt_schar))]) : bijective?(f)
```

The semantics of `unsigned char` is very similar to the one of signed characters. It only fixes the value type (like all other unsigned types). For the semantics of `char` we add the following to the usual setup:

```
signed_or_unsigned : Axiom
  (semantics_char_pred = semantics_uchar_pred) Xor
  (semantics_char_pred = semantics_schar_pred)

char_same_size : Axiom
  (semantics_char_pred = semantics_uchar_pred Implies size(dt_char) = size(dt_uchar)) And
  (semantics_char_pred = semantics_schar_pred Implies size(dt_char) = size(dt_schar))
```

These axioms require that the value type of `char` coincides with either `unsigned char` or `signed char`. Further the object representation must have the right size. With these assumptions it is possible to prove in PVS that also `char` has the ability to copy POD objects:

```
dt_char_from_byte_total: Lemma Forall (l: list[Byte], a : Address):
  length(l) = size(dt_char) Implies up?(from_byte(dt_char)(l, a))

dt_char_injective: Lemma Forall (l: list[Byte], a : Address):
  length(l) = size(dt_char) Implies to_byte(dt_char)(down(from_byte(dt_char)(l, a)), a) = l
```

The proof is not trivial: Consider the function to_byte and fix the second argument of type Address. The data_type? predicate implies that this function is injective with a finite codomain (byte lists of a fixed length). Therefore also its domain, the value type of `char`, must be finite. The Lemma schar_iso and the two axioms for `char` imply that the value type and the object representation of `char` have the same cardinality. So the to_byte function (with fixed second argument) must also be surjective. The preceding two lemmas follow now because from_byte is a left inverse of to_byte.

4.5 Void

The type `void` is very special. In C++ it is used as return type for functions that only produce side effects. Besides that, any value can be converted into type `void`, so there are expressions of type `void`. Nevertheless the standard specifies `void` as an *empty* type (§3.9.1.(9)). For a set-theoretic interpretation all this does not make much sense.

For the semantics of `void` we see the following alternative:

- Model `void` as the empty type (which does exist in PVS). In this case all functions and conversions with a codomain of `void` must be specially treated such that they do not create inhabitants in the empty type.
- Model `void` as an one-element type. In this case it is difficult to stay consistent with most C++ implementations, which optimise away any value of type void. One possible way is to allow that the only inhabitant of `void` does not occupy any memory.

We opt for the second alternative and explicitly permit models of `void` with empty object representation (i.e., size(dt_void) = 0).

4.6 Standard Conversions

The formalisation of the standard conversions is straigthforward. Here we show only three examples:

```
cnv_sc2b(sc : Semantics_schar) : Semantics_bool = sc /= 0
cnv_b2sc(b : Semantics_bool) : Semantics_schar = IF b Then 1 Else 0 Endif

cnv_uc2sc(uc : Semantics_uchar) : Semantics_schar
cnv_uc2sc_prop: Axiom Semantics_schar_pred(uc) Implies cnv_uc2sc (uc) = uc
```

The first two conversions from `signed char` to `bool` and back are completely specified in §4.7 (4) and §4.12 (1). Therefore we define the semantics of these conversions as functions. The behaviour of the third conversion from `unsigned char` to `signed char` is only partially specified in §4.7 (3), so we use a declaration together with an axiom.

5 Structures

Recall that we aim at a shallow embedding of C++ in PVS. Therefore we cannot formalise the semantics of all structures or all union types. We can only describe rules how to generate a semantics for a specific compound type. These rules will be compositional in the sense that the semantics of a compound type T that contains a member of type T_m, relies on the semantics of T_m. Our semantics compiler will implement these rules and generate a semantics for every compound type in the source. In this section we describe the semantics of structures. The other compound types remain future work, see Section 6.

The C++ version of the cartesian product is called `struct`. A structure combines several members of different types. Here is a typical example:

```
struct Z { int x; char y; };
```

It defines a new type Z. Any object of that type contains an integer and a character. Both can be accessed independently. There are several specific things for C++ structures: First one can access the whole structure or individual fields. The access to one field is independent of whether the other fields are initialised or not. It is perfectly legitimate to work with partly initialised structures, as long as one accesses only the initilized fields. Further, structures may contain padding, that is, there might be unused memory between any two fields of a structure (for instance, to satisfy alignment requirements). However, for POD structures, there is no padding at the beginning (§9.2 (17)). If there is no intervening access specifier the members are allocated in the same order as they are declared in the source code.

Note that classes in C++ are structures with a different default for the access specifier (**private** for classes, **public** for structures). In the semantics we do not distinguish between classes and structures.

In the following we present the semantics of the example structure Z. The general semantics for structures becomes clear from that. First we define the value type for the structure Z as a record of its member types. Next we define a second record for keeping the offsets of the individual members in the structure:

```
Semantics_Z : Type = [# x: Semantics_int, y: Semantics_char #]
Offsets_Z : Type =    [# x_offs: nat, y_offs: nat #]
```

Next we have the usual declarations for the common data type structure together with a constant offsets_Z that provides the indices of the members in the object representation of Z:

```
dt_Z_exists : Axiom Exists (x: (pod_data_type?[Semantics_Z])): true
dt_Z : (pod_data_type?[Semantics_Z])
offsets_Z : Offsets_Z
```

The semantics is completed with three axioms. The first one describes the order of the indices in offsets_Z and requires that the individual objects in the structure do not overlap:

```
offs_order_Z: Axiom 0 = x_offs(offsets_Z) And
  x_offs(offsets_Z) + size(dt_int) <= y_offs(offsets_Z) And
  y_offs(offsets_Z) + size(dt_char) <= size(dt_Z)
```

If the declaration of the structure contains access specifiers, then this axiom changes slightly and fixes only the order of members that are not separated by an access specifier (§9.2 (12)).

The second axiom states that one can access the individual fields if one can access the whole structure. Further, it does not matter whether one accesses one field or accesses the whole structure and selects that field in the value type.

```
from_byte_whole_Z: Axiom
  Forall(bl: list[Byte], a : Address):     up?(from_byte(dt_Z)(bl,a)) Implies
    Let x = down(from_byte(dt_Z)(bl,a)),
        lift_a = from_byte(dt_int)(sublist(x_offs(offsets_Z), size(dt_int))(bl),
                                   a + x_offs(offsets_Z)),
        lift_b = from_byte(dt_char)(sublist(y_offs(offsets_Z), size(dt_char))(bl),
                                    a + y_offs(offsets_Z))
    IN
        up?(lift_a) And down(lift_a) = x(x) And
        up?(lift_b) And down(lift_b) = y(x)
```

The function sublist(offs, len)(l) cuts out the sublist of length len of l that starts at offs. The third axiom is kind of inverse: If one can access all fields then one can access the whole structure and get the expected result:

```
from_byte_parts_Z: Axiom
  Forall(bl: list[Byte], a : Address):     length(bl) = size(dt_Z) Implies
    Let lift_x = from_byte(dt_Z)(bl, a),
        lift_a = from_byte(dt_int)(sublist(x_offs(offsets_Z), size(dt_int))(bl),
                                                    a + x_offs(offsets_Z)),
        lift_b = from_byte(dt_char)(sublist(y_offs(offsets_Z), size(dt_char))(bl),
                                                     a + y_offs(offsets_Z))
    IN
      up?(lift_a) And up?(lift_b) Implies
        up?(lift_x) And down(lift_a) = down(lift_x)`x And down(lift_b) = down(lift_x)`y
```

To prove consistency one can use a model that stores all members one after each other in one byte string of sufficient length. The required proofs are easy once one has the appropriate rewrite lemmas about sublist.

In this simple example we have neglected a few issues because their treatment is rather obvious: Inheritance of structures becomes a substructure hierarchy in the semantics. If sharing occurs (virtual base classes) one has axioms that state that different substructures are identical. Static member functions are not part of the structure, they are modelled as part of the program. The semantics of a structure with virtual member functions contains an additional field that holds its dynamic type. This field is used for the semantics of late binding. The details will be described in [10].

6 Future Work

In this section we remark on those parts of the semantics that have not been fully worked out yet.

Unions Unions are like structures with the important difference that all members are stored in it with offset zero. So in a union only one of its members can be active at any time. For the semantics of unions we need to express the value type of a union in the PVS type system. For this we cannot use disjoint unions because in the usual model of unions one cannot decide which member is active. The construction of a type constructor that is suitable for the value types remains future work.

Pointers The standard guarantees only very few things about pointers. One can, for instance, convert any pointer into a void pointer and back, without loosing information. For all the other pointer conversions the standard only requires that they preserve the null pointer. For the efficient implementation of an operating system one needs much more properties. For the verification of Fiasco it makes sense to identify the address type of the memory abstraction with the value type of all pointer types. In this case the semantics cannot detect if a pointer moves past the array bounds. For a general semantics of C++ it would be nice to permit also smart pointers as models of the pointer types. A smart pointer keeps information about its associated array and its type to detect errors of pointer arithmetic. To permit smart pointers it is necessary to leave the value type for pointers open.

Constant Objects In C++ one can qualify any object as constant. Intuitively a const object should never be changed after its initialisations (but compare the open issue 290 in [1]). The general setup of our semantics cannot faithfully model constant objects. There are the following two possible solutions.

- The semantics compiler rejects any program that casts a constant type into a non-constant type. For the remaining program one can check statically if they treat constant objects correctly.
- One enriches the memory interface to permit an operation that changes memory areas to read-only at runtime. In this case the semantics is able to detect an attempt to write to a constant object. One can then verify programs that need to cast away the `const` modifier because some arguments of a library function are (erroneously) not declared as constant.

For the VFiasco project the first solution is suffient.

Alignment Alignment describes the inability of some CPU's to access all data on all possible addresses. For instance on a sparc architecture a 2 byte memory access is only possible on even addresses. The x86 architecture has no alignment requirements. The framework presented in this paper does not support alinment requirements. An easy way to support alignment is to make the `to_byte` function partial. It can then fail if alignment requirements are not met. At the moment it is not clear to us how to model alignment requirements such that it applies to all possible architectures.

Float and Double The standard says almost nothing about the floating point types (apart from that they exist). For the VFiasco project the floating point types are not necessary. For the verification of C++ floating point programs one needs more assumptions than the

standard provides. For this one can reuse one of the several floating point formalisations, for instance [13].

Miscellaneous A complete semantics for the C++ data types must also treat references, bit fields, and volatile objects. For the VFiasco project only bit fields are relevant.

7 Conclusion

In this paper we describe a semantics of the data types of the C++ programming language. The semantics can deal with dynamically allocated objects, pointer conversion, and even with typing errors. The semantics can easily be adjusted to model additional features of a specific C++ implementation. This paper reports on work-in-progress. Here, we describe the general setup, the semantics of the fundamental or builtin types, and that of structures. We only comment on the other data type constructions.

References

[1] JTC1/SC22 Working Group 21. C++ standard core language active issues, revision 26, April 2003. available via http://std.dkuug.dk/JTC1/SC22/WG21/.

[2] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer, Berlin, 1991.

[3] R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, Edinburgh, Scotland., 1972.

[4] Shigeru Chiba. A metaobject protocol for C++. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 285–299, October 1995.

[5] International Organization for Standardization. *ISO/IEC 14882:1998: Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, September 1998.

[6] Y. Gurevich and J. Huggins. The Semantics of the C Programming Language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *LNCS*, pages 274–309. Springer, 1993.

[7] H. Härtig, R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS: OS support for distributed multimedia applications. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[8] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *16th ACM Symposium on Operating System Principles (SOSP)*, pages 66–77, Saint-Malo, France, October 1997.

[9] M. Hohmuth and H. Härtig. Pragmatic nonblocking synchronization for real-time systems. In *USENIX Annual Technical Conference*, Boston, MA, June 2001.

[10] M. Hohmuth and H. Tews. The C++ object model: A semantics for late binding and pure virtual functions. In preparation.

[11] M. Hohmuth and H. Tews. A semantics for C++ statements: Formalising harmful goto's, Duff's device and other monstrosities. In preparation.

[12] M. Hohmuth, H. Tews, and S. G. Stephens. Applying source-code verification to a microkernel — the VFiasco project (extended abstract). In *Proceedings of the Tenth ACM SIGOPS European Workshop*, September 2002.

[13] C. Jacobi. Formal verification of a theory of ieee rounding. In R.J. Boulton and P.B. Jackson, editors, *TPHOLs 2001: Supplemental Proceedings*, Informatics Research Report EDI-INF-RR-0046, Edinburgh, UK, 2001.

[14] B. Jacobs. Java's integral types in pvs, 2003. Manuscript.

[15] F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. In F. Baader, editor, *Automated Deduction — CADE-19*, Lecture Notes in Computer Science. Springer, 2003.

[16] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[17] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer, Berlin, 1996.

[18] S. Owre and N. Shankar. Theory interpretations in pvs. Technical Report SRI-CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park, CA, April 2001.

[19] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference, Version 2.4*. SRI International, Menlo Park, CA, December 2001.

[20] B. Stroustrup. *The C++ Programming Language: Third Edition*. Addison-Wesley Publishing Co., Reading, Mass., 1997.

[21] H. Tews, H. Härtig, and M. Hohmuth. VFiasco — towards a provably correct μ–kernel. Technical Report TUD-FI01-1 – January 2001, Dresden University of Technology, Department of Computer Science, 2001. Available via http://wwwtcs.inf.tu-dresden.de/∼tews/science.html.

[22] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312, 2001.

[23] C. Wallace. The Semantics of the C++ Programming Language. In E. Börger, editor, *Specification and Validation Methods*, pages 131–164. Oxford University Press, 1995.

Modeling and Verification of Leaders Agreement in the Intrusion-Tolerant Enclaves Using PVS

Mohamed Layouni[1], Jozef Hooman[2] and Sofiène Tahar[1]

[1] Department of Electrical and Computer Engineering,
Concordia University, Montreal, Canada
{layouni,tahar}@ece.concordia.ca
[2] Computing Science Department,
University of Nijmegen, Nijmegen, The Netherlands
hooman@cs.kun.nl

Abstract. Enclaves is a group-oriented intrusion-tolerant protocol. Intrusion-tolerant protocols are cryptographic protocols that implement fault-tolerance techniques to achieve security despite possible intrusions at some parts of the system. Among the most tedious faults to handle in security are the so-called Byzantine faults, where insiders maliciously exhibit an arbitrary (possibly dishonest) behavior during executions of the protocol. This class of faults poses formidable challenges to current verification techniques and has been formally verified only in simplified forms and under restricted fault assumptions. In this paper we present our work on the formal verification of the Byzantine fault-tolerant Enclaves [1] protocol. We use PVS to formally specify and prove Proper Byzantine Agreement, Agreement Termination and Integrity.

1 Introduction

We have seen in the last decade a substantial progress in the formal verification of cryptographic protocols. A wide variety of techniques have been developed to verify a number of key security properties ranging from *confidentiality*, *authentication* to *atomic transactions* and *non-repudiation* [2–5]. Nevertheless, all the focus was either on two-party protocols (i.e. involving only a pair of users) or, in the best cases, on group protocols with centralized leadership (i.e. a presumably trusted fault-free server managing a group of users). In the present work, we are concerned with the verification of the intrusion-tolerant Enclaves [1]: a group-membership protocol with a distributed leadership architecture, where the authority of the traditional single server is shared among a set of n independent elementary servers, f of which at most could fail at the same time. The protocol has a maximum resilience of one third (i.e. $f \leq \lfloor \frac{n-1}{3} \rfloor$) and uses a similar algorithm to the consistent broadcast of Bracha and Toueg [8].

The primary goal of Enclaves is to preserve an acceptable group-membership service of the overall system despite intrusions at some of its subparts. For instance, an authorized user u who requests to join an active group of users should be eventually accepted, despite the fact that faulty leaders may coordinate their

messages in such a way as to mislead non-faulty leaders (the majority) into disagreement, and thus into rejecting user u.

To achieve its intrusion-tolerant capabilities, Enclaves relies on the combination of a cryptographic authentication protocol, a Byzantine fault-tolerant leader agreement protocol and a secret sharing scheme. Although we assume the underlying cryptographic primitives and fault-tolerant components to be perfect, one cannot easily guarantee security of the whole protocol. In fact, several protocols had been long thought to be secure until a simple attack was found (see [20] for a survey). Therefore, the question of whether or not a protocol actually achieves its security goals becomes paramount. To date, most of the research in protocol analysis has been devoted to finding attacks on known, either two-party or centralized protocols. In this paper we are concerned with the verification of a distributed multi-leader group communication protocol.

Enclaves is intended to tolerate Byzantine faults [7]. Modeling Byzantine behavior has been always a big issue in formal verification. It arises the problem of how much power should be given to a Byzantine fault and how general the model should be to capture the arbitrary nature of a Byzantine fault behavior. These questions have been extensively studied [11–13] and continue to be a center of focus. In this paper faults are only limited by cryptographic constraints. For instance, they can arbitrarily send random messages, reset their local clocks and perform any action without satisfying its preconditions. Faults, however, cannot decrypt a message without having the appropriate key, or impersonate other participants by forging cryptographic signatures. More details about our fault assumptions are discussed in Section 2.

In this paper is we discuss a formal analysis of the Byzantine fault-tolerant leaders agreement module used of Enclaves. This module relies, to a large extent, on the timing and the coordination of a set of distributed actions, possibly performed by faulty processes whose behavior is hard to assess in any automatic verification tool. Therefore, we found it more convenient to proceed by means of theorem proving. In fact, we use PVS [14] and formalize the protocol in the style of Timed-Automata [9]. This formalism makes it easy to express timing constraints on transitions. It also captures several useful aspects of real-time systems such as liveness, periodicity and bounded timing delays. Using this formalism, we specified the protocol for any instance of size n, and we proved safety and liveness properties such as Proper Agreement, Agreement Termination and Integrity.

The remainder of this paper is organized as follows. In Section 2, we give an overview of the Enclaves protocol architecture and goals, and we explicitly state our system model assumptions. In Section 3, we present how we model the elementary components of the Byzantine leader agreement module in PVS and how we build the final protocol model out of these ingredients. In Section 4, we formulate and prove our theorems. In Section 5, we discuss some related work. Finally in Section 6, we comment our results and state some perspectives for a future work.

2 Overview on the Enclaves Protocol

Enclaves [1] is a protocol that enables users to share information and collaborate securely through insecure networks such as the Internet. Enclaves provides services for building and managing groups of users. Access to a given group is granted only to sets of users who have the right credentials to do so. Authorized users can dynamically and at their will join, leave, and rejoin an active group.

The group communication service relies on a secure multicasting channel that ensures integrity and confidentiality of group communication. All messages sent by a group member are encrypted and delivered to all the other group members.

The group-management service consists of user authentication, access control, and group-key distribution. Figure 1 shows the different phases of the protocol execution. Initially at time t_0, user u sends requests to join the group to a set of leaders. These leaders locally authenticate u within time interval $[t_1, t_2]$. When done, the agreement procedure starts and terminates at time t_4 by reaching a consensus as whether or not to accept user u. Finally on acceptance, user u is provided with the current group composition, as well as the group-key. Once in the group, each member is notified when a new user joins or a member leaves the group in such a way that all members are in possession of a consistent image of the current group-key holders.

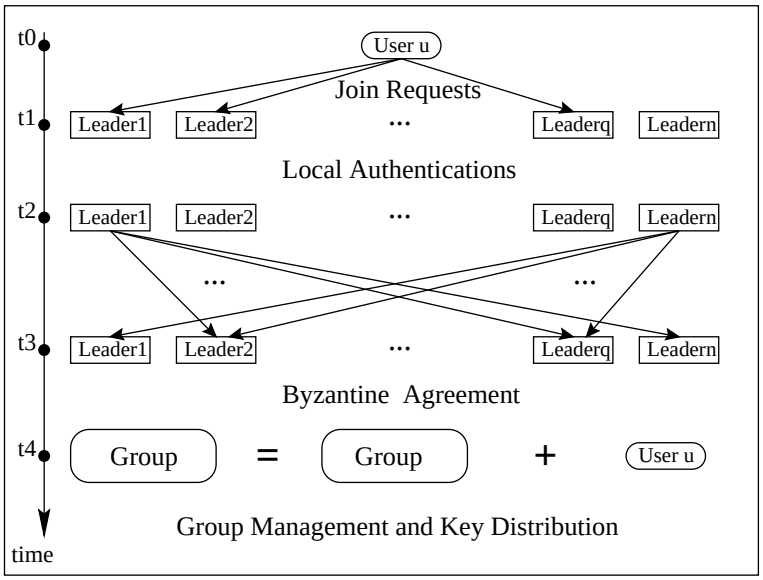


Fig. 1. Protocol execution

In summary, we prove that Enclaves satisfies the *Proper authentication and access control* requirement even in the presence of f compromised leaders. The latter requirement states that only authorized users can join the application and an authorized user cannot be prevented from joining the application. This has been established in PVS through the Proper Agreement, Agreement Termination and Integrity theorems (Sections 3).

The description of Enclaves in [1] assumes a reliable network where messages eventually reach their destinations within an upper bound delivery time. In this paper we make the same assumptions. Concerning the intruder, we adopt a standard model where an intruder fully monitors the network, proactively augments its knowledge, and chooses to send, either adaptively or randomly, messages on the network. The intruder, however, cannot block messages from reaching their destination and is limited by cryptographic constraints. For instance, the intruder cannot decrypt messages without having the right key, or impersonating other participants by forging cryptographic signatures. Given the above settings, we assume the cryptography layer to be perfect (i.e. messages format is well chosen to prevent any leakage of sensitive information), and we concentrate rather on the Byzantine fault-tolerance capabilities of the protocol.

Next, we formalize the elementary components of the Byzantine leader agreement module in PVS and we build the final protocol model out of these ingredients. Then in Section 4, we formulate and prove our theorems.

3 Modeling Intrusion-Tolerant Enclaves in PVS

Most group communication protocols, including Enclaves, can be modeled by an automaton whose initial state is modified by the participants' actions as the group mutates (e.g., new members join). Because Enclaves depends also on time (participants timeout, timestamp group views etc.), it is natural to model it as a timed automaton. Participants in a typical run of Enclaves consist of a set of n leaders (f of which are faulty), a group of members, and one or more users requiring to join the group. Similarly to the PAXOS protocol in [16], the leaders communicate with each others and with users via a partially asynchronous network. Messages sent on this network are assumed to be eventually delivered to their destinations within an upper bound of time, but no assumption is made on the reception order.

In the remainder of this section, we first explain our general PVS theory about timed automata. The parameters of this theory are used here to formalize Enclaves by defining the actions, the states, and the preconditions and effects of each action. Finally, the resulting executions of the protocol and fault assumptions are described.

3.1 Timed Automata Model in PVS

We present a general, protocol-independent, theory called `TimedAutomata`. Given a number of parameters, which represent the essential characteristics of a protocol, it defines all possible executions of the protocol as a set of `Runs`. A *run* is a sequence of the form $s_0 \stackrel{a_0}{\rightarrow} s_1 \stackrel{a_1}{\rightarrow} s_2 \stackrel{a_2}{\rightarrow} s_3 \stackrel{a_3}{\rightarrow} \ldots$ where the s_i are *states*, representing a snapshot of the system during execution and the a_i are the executed *actions*. A particular protocol is characterized by sets of possible `States` and `Actions`, a condition `Init` on the initial state, the precondition `Pre` of each state, expressing which states can be executed, the effect `Effect` of each action, expressing the possible state changes by the action, and a function `now` which gives the current time in each state. In a typical application, there is a special *delay* action which models the passage of time and increases the value of `now`. All other actions do not change time. In PVS, the theory and its parameters are defined as follows[1].

```
TimedAutomata  [ States, Actions: TYPE+,
                 Init : pred[States],
                 Pre : [Actions -> pred[States]] ,
                 Effect  :  pred[[States, Actions, States]],
                 now : [States -> nonneg_real]
               ] : THEORY
```

To define runs, let `PreRuns` be a record with two fields, `states` and `events`.

```
PreRuns : TYPE = [# states  : sequence[States],
                    events  : sequence[Actions] #]
```

A `Run` is a `PreRun` where the first state satisfies `Init`, the precondition and effect predicates of all actions are satisfied, the current time never decreases and increases above any arbitrary bound (avoiding Zeno-behaviour [6]). In PVS this is formalized as follows.

```
PreEffectOK(pr) : bool = FORALL i :
   Pre(events(pr)(i)) (states(pr)(i)) AND
   Effect(states(pr)(i), events(pr)(i), states(pr)(i + 1))

NoTimeDecrease(pr) : bool =
   FORALL i : now(states(pr)(i)) <= now(states(pr)(i + 1))

NonZeno(pr): bool =
   FORALL t :   EXISTS i :  t < now(states(pr)(i))

Runs : TYPE =
   { pr: PreRuns | Init(states(pr)(0)) AND PreEffectOK(pr) AND
                   NoTimeDecrease(pr) AND NonZeno(pr) }
```

[1] More details about the PVS theories and proofs can be found at the project web page: http://hvg.ece.concordia.ca/Publications/TECH_REP/PVS_TR03/PVS_TR03.html

3.2 Leaders Actions

To define the actions of the leaders, we first state a few preliminary definitions. Let n be the number of leaders and let f be such that $3f + 1 \leq n$ (the maximum number of faulty leaders). For simplicity, leaders are identified by an element of $\{0, 1, \ldots, n-1\}$. Users are represented by some uninterpreted non-empty type. We model time as a non-negative real number and define three time constants for the maximum delay of messages in the network, the maximum delay between trypropagate actions and the maximum delay between tryaccept actions. Details are shown below:

```
n  : posnat
f : { k : nat | 3 * k + 1 <= n }

LeaderIds : TYPE = below[n]
UserIds   : TYPE+
Time :      TYPE+ = nonneg_real

i       : VAR LeaderIds
user    : VAR UserIds
t       : VAR Time
MaxMessageDelay, MaxTryPropagate, MaxTryAccept  : Time
```

The actions of the protocol are represented in PVS as a data type, which ensures, e.g., that all actions are syntactically different.

```
LeaderActions [LeaderIds, UserIds, Time : TYPE] : DATATYPE
BEGIN
   delay(del : Time) : delay?
   announce(id : LeaderIds, user : UserIds) : announce?
   trypropagate(id : LeaderIds) : trypropagate?
   tryaccept(id : LeaderIds) : tryaccept?
   receive(id : LeaderIds) : receive?
   crash(id : LeaderIds) : crash?
   misbehave(id : LeaderIds) : misbehave?
END LeaderActions
```

Informally, these actions have the following meaning:

- *delay* is a general action which occurs in all our timed models; it increases the current time (`now`), and all other clocks that may be defined in the system, with the amount specified by parameter `del`.
- The *announce* action is used to send announcement messages of new locally authenticated users to the other leaders of the protocol.
- The *trypropagate* action allows a user announcement to be further spread among leaders. This action is executed periodically, but it only changes the state of the system if enough announcements $(f + 1)$ have been received for the considered user and it has not already been announced or propagated by the leader in question before.

- *Tryaccept* is used to let leaders periodically check whether they have received enough many announcements and/or propagation messages for a given user. Once this condition is satisfied, the user is accepted as a member of the group.
- The *receive* action allows a leader to receive messages. More concretely, it is used to remove a received message from the network and to add corresponding data to the leaders local buffers.
- The *crash* action models the failure of a leader. After a crash it may still perform all the actions mentioned above, but in addition may perform a *misbehave* action.
- Action *misbehave* models the Byzantine mode of failure and can only be performed by a faulty (crashed) leader.

3.3 States

In order to properly capture the distributed nature of the network, it is suitable to model two kinds of states: a local state for each leader, accessible only to the particular leader, and a global state to represent global system behavior which includes the local state of each leader, the representation of the network and a global notion of time.

An important part of the local state is the group `views`, which is a set of users in the current group. In fact, the ultimate goal of Enclaves is to assure consistency of the group views. Moreover we have a boolean flag `faulty` marking the leader status as to faulty or not, some local timers `clockp` and `clocka` to enforce upper bounds on the occurrence of `trypropagate` and `tryaccept` actions, and finally `received`, a list of the leaders from which the local leader received proposals for a given user.

```
Views : TYPE = setof[UserIds]

LeaderStates : TYPE =
  [# view          : Views,
     faulty        : bool,
     clockp        : Time,   % clock for the trypropagate action
     clocka        : Time,   % clock for the tryaccept action
     received      : [UserIds -> list[LeaderIds]]   #]
```

We model `Messages` as quadruples containing a source, a destination, a proposed user and a timestamp indicating an upper bound on the delivery time, i.e. the message must be received before the `tmout` value.

```
Messages : TYPE = [# src      : LeaderIds,
                     tmout    : Time,
                     proposal : UserIds,
                     dest     : LeaderIds   #]
```

In the `GlobalStates` the network is modeled as a set of messages. Messages that are broadcast by leaders are added to this set, with a particular time-out

value, and they are eventually received, possibly with different delays and at a different order at recipient ends. The global state also contains the local state of each leader and a global notion of time, represented by `now`.

```
GlobalStates : TYPE = [# ls      : [LeaderIds -> LeaderStates],
                         now     : Time,
                         network : setof[Messages]  #]
s, s0, s1   : VAR GlobalStates
```

Predicate `Init` expresses conditions on the initial state, requiring that all views, received sets and the network are empty, all clocks and now are zero.

3.4 Precondition and Effect

For each action `A` we define its precondition, expressing when the action is enabled, and its effect. An `announce` action may always occur and hence has precondition `true`. Similarly for `trypropagate` and `tryaccept` which should occur periodically. Action `receive(i)` is only allowed when there exists a message in the network with destination `i`. For simplicity, a `crash` action is only allowed if the leader is not faulty (alternatively, we could take precondition `true`). A `misbehave` action may only occur for faulty leaders.

Most interesting is the precondition of the `delay(t)` action. This action increases `now` and all timers `clockp` and `clocka` by `t`. To ensure that messages are delivered before their time-out value, we require that condition `prenetwork` holds in the state before a `delay(t)` action.

```
prenetwork(s, t) : bool =  FORALL msg :
    member(msg, network(s)) IMPLIES  now(s) + t <= tmout(msg)
```

Note that when the time-out of a message equals the current time, no delay action is possible; first a receive action has to occur. Similarly, there is a condition `preclock` which requires that all timers `clockp` and `clocka` are not larger than `MaxTryPropagate` and `MaxTryAccept`, respectively. Since the `trypropagate` and `tryaccept` actions reset their local timers, `clockp` and `clocka` resp., to zero, this may enforce the occurrence of such an action before a time delay is possible.

```
Pre(A)(s) : bool =
   CASES A OF
     delay(t)        : prenetwork(s,t) AND preclock(s,t),
     announce(i,u)   : true,
     trypropagate(i) : true,
     tryaccept(i)    : true,
     receive(i)      : MessageExists(s,i),
     crash(i)        : NOT faulty(ls(s)(i)),
     misbehave(i)    : faulty(ls(s)(i))
    ENDCASES
```

Next we define the effect of each action, relating a state `s0` immediately before the action and a state `s1` immediately afterwards.

- `delay(t)` increments `now` and all local timers by `t`, as defined by `s0+t`.
- `announce(i,u)` adds, for each leader `j` a message to the network, with source `i`, time-out `now(s0)+MaxMessageDelay`, proposal `u`, and destination `j`. This is expressed by `AnnounceEffect(s0,i,u,s1)`.
- `trypropagate(i)` resets `clockp` to zero and adds to the network messages, to all leaders, containing proposals for each user for which at least `f+1` messages have been received.
- `tryaccept(i)` resets `clocka` to zero and adds to its local view all users for which at least `n-f` messages have been received.
- `receive(i)` removes a message with destination `i` from the network, say with source `j` and proposal `u`, and adds `j` to the list of received leaders for `u` provided it is not in this list already.
- `crash(i)` sets the flag `faulty` of `i` to `true`.
- `misbehave(i)` may just reset the local timers `clockp` and `clocka` of `i` to zero (and hence it need not try to propagate or accept periodically), as expressed by `ResetClock(s0,i,s1)`, or it may add randomly, and above all, maliciously chosen messages to the network (as long as the timing constraints are not violated). A faulty leader, however, cannot impersonate other protocol participants, i.e., cannot send messages on behalf of other participants and any message sent on the network has the identifier of its actual sender.

This leads to a predicate of the following form:

```
Effect(s0,A,s1) : bool =
   CASES A OF
     delay(t)        : s1 = s0 + t,
     announce(i,u)   : AnnounceEffect(s0,i,u,s1),
     trypropagate(i) : PropagateEffect(s0,i,s1),
     tryaccept(i)    : AcceptEffect(s0,i,s1),
     receive(i)      : ReceiveEffect(s0,i,s1),
     crash(i)        : CrashEffect(s0,i,s1),
     misbehave(i)    : ResetClock(s0,i,s1) OR SendMessage(s0,i,s1)
    ENDCASES
```

3.5 Protocol Runs and Fault Assumption

Runs of this timed automata model of Enclaves are obtained by importing the general timed automata theory. This leads to type `Runs`, with typical variable `r`.

```
IMPORTING TimedAutomata[GlobalStates, LeaderActions, Init,
                        Pre, Effect, now]
r           : VAR Runs
```

Let `Faulty(r,i)` be a predicate expressing that leader `i` has a state in which it is faulty. It is easy to check in PVS that once a leader becomes faulty, it remains faulty forever.

Let `FaultyNumber(r)` be the number of faults in run `r` (it can be defined recursively in PVS). Then we postulate, by an axiom with name `MaxFaults`, that the maximum number of faults is `f`.

```
MaxFaults : AXIOM FaultyNumber(r) <= f
```

4 Formal Verification

We verify the following properties of the Intrusion-Tolerant Enclaves protocol:

- **Termination**: if a user u wants to join an active group and is locally authenticated by a non-faulty leader, then user u will be eventually accepted by all non-faulty leaders and become a member of the group.
- **Integrity**: a user that has been accepted in the group should have been locally authenticated and announced by a non-faulty leader earlier during the protocol execution.
- **Proper Agreement**: if a non-faulty leader decides to accept a user u, then all non-faulty leaders accept user u too.

In the remainder of this section, we first present the proof of the termination property, and next give a short outline of the proof for integrity and finally derive proper agreement.

Theorem 1 (Termination)
For all `r` *and* `u`, `announced_by_many(r,u)` *implies* `accepted_by_all(r, u)`

where

- `announced_by_many(r,u)` expresses that at least $f + 1$ non-faulty leaders announced user `u` during run `r`;
- `accepted_by_all(r, u)` asserts that eventually all non-faulty leaders have user `u` in their `view` during run `r`.

Proof Assume `announced_by_many(r,u)`, which implies that at least $f + 1$ non-faulty leaders broadcast a proposal for `u`. Because of the reliability of the network, these messages will be eventually delivered to their destination, and in particular to the $n - f$ non-faulty leaders of the network. They all receive $f + 1$ announcement messages for user `u`, enough to have `u` in their `PropSets` and trigger the propagation procedure for all non-faulty leaders who did not participate in the announcement phase. Now because of the network reliability, we conclude that eventually all non-faulty leaders will receive at least $n - f$ approvals for user u, enough to make a majority ($n - f > f$ as $n > 3f$). □

The formal proof of this theorem in PVS required over 25 intermediate lemmas, most of which are of average difficulty. The proof backbone resides on the following steps (which are represented as lemmas in PVS):

- `announced_by_many(r,u)` implies that eventually user `u` has been received by all non-faulty leaders at least $f + 1$ times.
  Note that this requires the use of the `NonZeno` property of runs, to obtain a state with global time greater than the largest time-out value in the messages; then, by the precondition of the `delay` action, all messages must have been received.
- If all non-faulty leaders have received proposals for `u` at least $f + 1$ times then all non-faulty leaders eventually broadcast a proposal for `u`.
  In this step the `NonZeno` property is used to reach a state where time has advanced more than `MaxTryPropagate` time units. Hence, given the condition on `clockp` for a `delay` action, a `trypropagate` action should have happened before.
- If all non-faulty leaders eventually broadcast a proposal for `u` then all non-faulty leaders eventually receive a proposal for `u` from all non-faulty leaders.
  Again this follows from NonZenoness and the precondition of the `delay` action.
- If all non-faulty leaders eventually receive a proposal for `u` from all non-faulty leaders, then they all have received at least $n - f$ proposals for `u`.
- If all non-faulty leaders eventually have received at least $n - f$ proposals for `u` then they all eventually accept `u`, i.e. include `u` in their `view`.
  Here we need NonZenoness and the condition on `clocka` for a `delay` action.

Details of the proof concern invariance properties about upper bounds of the local timers `clockp` and and `clocka`, and fact about the `received` list (e.g. that leaders are never removed from this list).

Theorem 2 (Integrity)
For all `r` *and* `u`, `accepted_by_one(r,u)` *implies* `announced_by_one(r,u)`

where

- `accepted_by_one(r,u)` holds if at least one leader eventually included `u` in its `view` during run `r`.
- `announced_by_one(r,u)` expresses that at least one non-faulty leader announced user `u` during run `r`;

Proof We proceed by contrapositive and use the non-impersonation property. We assume that for all non-faulty leaders no announcement for user `u` has been done during run `r`. Now because of non-impersonation, faulty leaders cannot send more than f different announcements. This implies that the leaders would receive no more than f announcements for user `u`, which is not enough to trigger propagation actions. This yields that `u` will never be in any of the non-faulty leaders `PropSet`, and hence in none of the `AcceptSets`. As a result user `u` will never be accepted by any of the non-faulty leaders. □

Theorem 3 (Proper Agreement)
For all `r` *and* `u`, `accepted_by_one(r,u)` *implies* `accepted_by_all(r,u)`

Proof `accepted_by_one(r,u)` implies that there exists one non-faulty leader that received at least $n - f$ approvals (i.e. announcements or propagation messages) for user `u`. Among these approvals, at least $n - 2f$ come from non-faulty leaders (by non-impersonation). Now because these leaders are non-faulty, they broadcast the same approval to all the other leaders. In addition, because of the network reliability, these messages are eventually delivered to destination. This implies that all $n - f$ non-faulty leaders receive eventually the above $n - 2f$ approvals. Since $n - 2f > f + 1$, all $n - f$ non-faulty leaders have user `u` in their `PropSet`. Now like in the proof of Termination, the latter implies the start of the propagation procedure, then the reception of at least $n - f$ approvals for user `u`, and finally the acceptance of `u` by all non-faulty leaders. □

The above proofs were conducted in PVS and required over 40 lemmas.

5 Related work

Much work has been done to formally verify security and fault-tolerance in distributed protocols. Some of the methods dealt with the Byzantine failure model while others remained limited to the benign form. Also some of them relied on specialized *security logics* (e.g., BAN [15], while others, remained more general and adopted a number of automata formalisms to prove protocols [11, 12, 16, 18].

Castro and Liskov [11] specified a Byzantine fault-tolerant replication algorithm (similar to ours) using the I/O automata of Tuttle and Lynch [10]. They have manually proved their algorithm's safety, but not its liveness, using invariant assertions and simulation relations. The work of Castro and Liskov is related to ours in the sense that it targets a Byzantine fault-tolerant multi-leader protocol. Although successful in pen-and-paper fashion, their work has never been conducted mechanically in a theorem prover.

Kwiatkowska and Norman [12] analyzed the Asynchronous Binary Byzantine Agreement [19] (based on a similar concept to our key management module) using a combination of mechanical inductive proofs (for non-probabilistic properties) and finite state checks (probabilistic properties) plus one high-level manual proof. Our approach too takes advantage of the easiness and performance of the different earlier mentioned techniques to prove the overall Enclaves protocol.

Lynch *et al.* used also timed automata to model their fault-tolerant protocols PAXOS [16] and Ensemble [21]. They assume a partially synchronous network and support only benign failures. This bears similarities with Enclaves verification in the sense that we also assume some bounds on the timing behavior, but unlike the work in [16, 21] we are dealing in this paper with the more subtle Byzantine form of failure.

In [18, 17], Heitmeyer and Archer presented the formal verification of the TESLA protocol using the Timed Automata Modeling Environment (TAME). TAME is a user interface to PVS. It provides a set of theory templates to specify general I/O automata. Our work can be used to extend the TAME package.

6 Conclusion and future work

Although formal verification techniques have reached a certain level of maturity, making complex and safety critical aspects of systems relatively easy to undertake, reasoning about systems involving Byzantine faults remained always a challenging task. In this paper we present our attempt to the formal specification and verification of the Byzantine agreement protocol used in the intrusion-tolerant Enclaves.

We believe we have achieved a promising success in verifying a complex protocol such as the Byzantine leaders agreement of Enclaves. Thanks to the high level of expressiveness of the Timed-Automata formalism, as well as the rich datatype package of PVS, we have succeeded to formalize the protocol for any instance of size n, in a way that thoroughly captures the different protocol subtleties. We have also proved the protocol to satisfy its requirements of *Termination*, *Integrity* and *Proper Agreement* under the earlier mentioned model and fault assumptions. The specification and proofs required respectively around 1200 lines of code and 40 intermediary lemmas, most of which are of average difficulty.

The current verification can be further extended by widening the Byzantine faults capabilities and by bringing, to the scene, the joint cryptographic layers yet abstracted away. This should make the model more complex, and might require a compositional verification of the different layers.

Acknowledgments

The formal specification and analysis of Enclaves benefited from the fruitful discussions with Adriaan DeGroot from University of Nijmegen.

References

1. Bruno Dutertre, Valentin Crettaz and Victoria Stavridou. Intrusion-Tolerant Enclaves. In: Proc. IEEE International Symposium on Security and Privacy, p. 216-226, Oakland, CA. May, 2002.
2. Catherine Meadows. The NRL Protocol Analyzer: An Overview. Journal of Logic Programming, 26(2):113-131,1996.
3. Peter Ryan and Steve Schneider. The Modelling and Analysis of Security Protocols: the CSP Approach. Addison-Wesley, 2000.
4. Lawrence C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. Journal of Computer Security, 6:85-128,1998.
5. Giampaolo Bella and Lawrence C. Paulson. Mechanical Proofs about a Non-Repudiation Protocol. In: Richard J. Boulton and Paul B.Jackson (editors), Theorem Proving in Higher Order Logics (LNCS 2152): p. 91-104, 2001.
6. Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, Sergio Yovine. Symbolic Model Checking for Real-time Systems. In: Proc. 7th. Symposium of Logics in Computer Science, Santa-Cruz, California, 1992.

7. Leslie Lamport, Robert Shostak and MARSHALL Pease. The Byzantine Generals Problem. In: ACM Transactions on Programming Languages and Systems, 4 (3), p.382-401, July 1982.
8. Gabriel Bracha and Sam Toueg. Resilient consensus protocols. In: Proceedings of the second annual ACM symposium on Principles of distributed computing, p.12-26, August 17-19, 1983, Montreal, Quebec, Canada
9. Rajeev Alur and David L. Dill. A Theory of Timed Automata. In Theoretical Computer Science 126: p.183-235, 1994.
10. Nancy Lynch. Distributed Algorithms. Morgan Kaufmann Publishers, 1996.
11. Miguel Castro and Barbara Liskov. A Correctness Proof for a Practical Byzantine-Fault-Tolerant Replication Algorithm. Technical Memo MIT/LCS/TM-590, MIT Laboratory for Computer Science, June 1999.
12. Marta Kwiatkowska and Gethin Norman. Verifying Randomized Byzantine Agreement. D.A. Peled, M.Y. Vardi (Eds.): Formal Techniques for Networked and Distributed Systems (LNCS 2529): p. 194-209, 2002.
13. Patrick Lincoln and John Rushby. A Formally Verified Algorithm for Interactive Consistency under a Hybrid Fault Model. In Fault Tolerant Computing Symposium, p. 304-313, Toulouse, France, June, 1993.
14. S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In 11th International Conf. on Automated Deduction, (LNCS 607): p. 748-752, 1992.
15. M. Burrows, M. Abadi and R. Needham. A Logic of Authentication, In: ACM Transactions on Computer Systems, 8(1): p. 18-36, 1990.
16. Roberto De Prisco, Butler W. Lampson, Nancy A.Lynch. Revisiting the PAXOS Algorithm. In Mavronicolas, M. and Tsigas, P., editors, 11th International Workshop on Distributed Algorithms, (LNCS 1320): p. 111-125, 1997.
17. Myla Archer, Constance Heitmeyer, and Elvinia Riccobene. Proving Invariants of I/O Automata with TAME. In Automated Software Engineering, Vol.9, p. 201-232, 2002.
18. Myla Archer. Proving Correctness of the Basic TESLA Multicast Stream Authentication Protocol with TAME. In Workshop on Issues in the Theory of Security, Portland, OR, January 14-15, 2002.
19. Christian Cachin, Klaus Kursawe and Victor Shoup. Random oracles in constantipole: practical asynchronous Byzantine agreement using cryptography (extended abstract). In Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing, p. 123-132, Portland, Oregon, 2000.
20. John Clark and Jeremy Jacob. A Survey of Authentication Protocol Literature: Version 1.0. Draft paper available at http://www-users.cs.york.ac.uk/~jac
21. Jason Hickey, Nancy Lynch, and Robbert van Renesse. Specifications and Proofs for Ensemble Layers. In 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, (LNCS 1579), p. 119-133, March, 1999.

∀UNITY : A HOL Theory of General UNITY

I.S.W.B. Prasetya[1], T.E.J. Vos[2], A. Azurat[1], and S.D. Swierstra[1]

[1] Institute of Information and Computing Sciences, Utrecht University.
email: {wishnu,ade,doaitse}@cs.uu.nl
[2] Instituto Tecnológico de Informática, Universidad Politécnica de Valencia.
email: tanja@iti.upv.es

Abstract. UNITY is a simple programming logic that can be used to reason about distributed systems. It is especially attractive because of its elegant axiomatical style. Its expresiveness is however limited, and for specific applications people often need variants of UNITY that offer extended or modified logic [23, 6, 27, 16, 21, 10, 20]. When modifying a logic mistakes easily creep in: a seemingly very logical new inference rule may turn out to be unsound. Formal verification is often necessary, but it is a time consuming task. ∀UNITY is a generalization that provides the same set of inference rules UNITY does. However, these inference rules are now derived from ∀UNITY's primitive rules that are weaker than the original ones. Consequently, a UNITY variant (or instance) can be created quickly just by proving that the instance upholds ∀UNITY primitive rules. The resulting UNITY variant is guaranteed to be sound since ∀UNITY is provided as a library within the HOL theorem prover, and all its derived rules are mechanically verified. Moreover, general application theories, such as theories about self-stabilization as in [19], can now be written based on ∀UNITY ; the theories will automatically extend to all ∀UNITY instances.

1 Introduction

UNITY is a programming language and an associated logic introduced by Chandy and Misra in 1988 to describe, specify and prove the correctness of distributed systems. The logic part consists of: (*a*) three operators, unless, ensures, and $\mapsto$ (leads-to), with which we can specify temporal properties of a distributed system; and (*b*) a set of inference rules to prove such properties. The logic is not really suitable for automated program verification, since $\mapsto$ properties needed to describe progress are not generally decidable[3]. However, UNITY is excellent for the formal treatment of distributed *algorithms* [12, 26, 3, 22, 15, 11, 5, 28, 31, 19, 29, 30], since these algorithms tend to be abstract, subtle, and parameterized with higher order information and thus also beyond the reach of automated verification. For this kind of usage, UNITY's simple and elegant axiomatic style is a big advantage.

UNITY's simplicity does mean that its expresiveness is limited. For example, it restricts itself to first order temporal properties. For many applications, UNITY is sufficiently expressive. Yet there are also applications which can be handled by a UNITY-like logic, which requires some modification or extension to the original UNITY, for example as in [23, 6, 27, 16, 21, 8, 10, ?,14][4]. However, when modifying a logic it is very easy to make a mistake (a seemingly very logical new inference rule

[3] Though in special cases, UNITY $\mapsto$ properties can be mapped to other formalisms where automated verification is possible.

[4] Some may object to spawning many variants of a logic. We do not share this view. Customizing a logic, just like reconfiguring a program, is driven by demand. When confronted with a new kind of problems, it makes sense to reconfigure existing logic to

may turn out to be unsound). There have been many examples of such mistakes, for example Sanders showed in 1991 [24] that the Substitution Law in UNITY [4] is actually unsound. It took several more years for people to realize that Sanders's own correction is also flawed [17]. Mechanical verification is therefore highly recommended when devising a new logic.

In 1992, Andersen successfully mechanized and verified the inference rules of UNITY using the theorem prover HOL [1]. Prasetya mechanized some variants of UNITY in HOL, including a variant that removes the flaw in Sanders UNITY [16]. Until now, to mechanically verify each new variant one will have to redo the entire proof, which is a costly and time consuming process.

∀UNITY is a generalization of UNITY. Although it provides the same set of inference rules, these are now derived from a set of more primitive (weaker) rules. Consequently, using ∀UNITY a UNITY variant can be easily created by only showing that the variant upholds ∀UNITY primitive rules. This is significantly less work than having to redo the entire UNITY soundness proof. ∀UNITY is provided as a library within the HOL theorem prover and all its derived laws have been mechanically verified. HOL ensures that any concrete UNITY variant which is derived from ∀UNITY is sound and complete, in the sense that it will satisfy all standard UNITY inference rules.

Effort has been made to make the derived rules of ∀UNITY minimal in the sense that each rule explicitly mentions which of the ∀UNITY primitive rules it requires. Consequently, it is possible to make an instantiation that, for example, only upholds the Completion Rule under certain circumstances.

∀UNITY is also useful for constructing general theories: any application theory which is based purely on ∀UNITY is also valid for all UNITY variants which can be instantiated from ∀UNITY . ∀UNITY can be downloaded from:

`www.cs.uu.nl/~wishnu/research/research.html`.

∀UNITY is defined in terms of generalized classical UNITY operators (unless and ensures) instead of the New UNITY [14] operators (co and transient). The choice is more a matter of taste: we stick with the classical ones because they are more familiar and intuitive. See also Appendix B about relation with New UNITY.

1.1 Contents of the paper

Section 2 starts by giving a brief review on classical UNITY and Section 3 explains the notation used in this paper. Subsequently, Section 4 shows some examples of instances of ∀UNITY, which then is presented in Section 5. Finally, Section 6 concludes.

Section 5 is somewhat informal. The formal definition of ∀UNITY 's primitive properties is listed in Appendix A. Due to limited space, we cannot show the complete list of inference rules derivable in ∀UNITY , but they can be found in the distribution package of ∀UNITY . Appendix B briefly comments on the relation with New UNITY.

2 Brief Review: UNITY

In UNITY, a distributed system is modeled by a set of actions, each of which is assumed to be atomic and terminating. A (concurrent) execution of such a system is an infinite and interleaved execution of its actions. In each step of the execution some action is non-deterministically selected from the set of enabled actions. Finite

be weakly fair, meaning that an action which is continually enabled (waiting to be executed) can not be ignored forever.

To specify the behavior of a program, three operators are provided: unless, ensures and $\mapsto$. Given two state predicates p and q, a program P is said to satisfy p unless q if: once p holds during an execution of P, it remains to hold at least until q holds. The program satisfies p ensures q if it satisfies p unless q and moreover there exists an action in P that can, and because of the fairness assumption of UNITY, will establish q. Their formal definitions are as follows:

Definition 1. : Unless and Ensures (Classical)

$$P \vdash p \text{ unless } q \;=\; (\forall a : a \in P : \{p \wedge \neg q\}\ a\ \{p \vee q\})$$

$$P \vdash p \text{ ensures } q \;=\; (P \vdash p \text{ unless } q) \;\wedge\; (\exists a : a \in P : \{p \wedge \neg q\}\ a\ \{q\})$$

Whereas unless specifies safety, ensures specifies progress. However, an ensures property can only specify progress which can be guaranteed by a single action. To describe progress in general, we use $\mapsto$. Informally, a program P satisfies $p \mapsto q$ if: whenever p holds during the execution of P then eventually q will also hold. Formally, $\mapsto$ is defined as the smallest transitive and left-disjunctive (i.e. at the p-position) closure of ensures.

To reason about the behavior of a program, the UNITY logic provides a set of inference rules. For example, one of the rules says that we can join two unless properties:

Theorem 1. : Unless General Conjunction (Classical)

$$\frac{P \vdash p \text{ unless } q \qquad P \vdash r \text{ unless } s}{P \vdash p \wedge r \text{ unless } (p \wedge s) \vee (r \wedge q) \vee (q \wedge s)}$$

As another example, the following rule states that if a program P can progress from p to q, and it can maintain a condition a at least until b holds, then starting from $p \wedge a$ it can either reach q while a still holds, or else then at least we know that it has entered the condition b. The rule is known as the *Progress Safety Progress* or PSP rule:

Theorem 2. : PSP (Classical)

$$\frac{P \vdash p \mapsto q \qquad P \vdash a \text{ unless } b}{P \vdash (p \wedge a) \mapsto (q \wedge a) \vee b}$$

For the complete list of UNITY rules see for example [4]. The rules (in the case of classical UNITY: without the Substitution law) are sound and in fact can be

3 Notation

We will use a notation that deviates from the usual UNITY style such as the one used in Section 2. Like UNITY, ∀UNITY is a formalism. However, it is implemented in the theorem prover HOL [9] as a HOL theory. Consequently, we will use the *HOL notation*, which is admittedly less stylish, but will make it easier for the reader to access the ∀UNITY HOL library.

Formulas are written in the type writer font and UNITY operators are written in the prefix-style, e.g. `UNLESS p q`. A UNITY inference rule will be written like this:

```
|- A1 /\ ... /\ An ==> C
```

which means that `C` is derivable from `A1` ... `An`. The notation makes an inference rule looks like an ordinary predicate logic theorem, which in fact it is: in ∀UNITY an inference rule is implemented as a HOL theorem (this also means that it is only a rule if its validity can be proven).

When defining a concrete UNITY (i.e. a ∀UNITY instance), we will write the UNITY operators with upper case letters, e.g. `UNLESS`, `ENSURES`, and `LEADSTO`. These upper case names refer to concretely defined objects. In ∀UNITY itself these operators are parameters and we will write them with lower case letters, e.g. `unless`, `ensures`, and `leadsto`.

There are two levels of logical operators in ∀UNITY . We have:

`/\`	(conjunction)
`\/`	(disjunction)
`==>`	(implication)
`~`	(negation)
`!`	(universal quantification)
`?`	(existential quantification)

with the usual meaning. Semantically, in HOL they are boolean operators. For example, `/\` takes two booleans and returns a boolean.

A state-predicate is a predicate like $x > y+1$ and is used to, for example, specify the set of possible states a program can be in at a given moment. The predicate can be simply represented by `x>y+1` in HOL. However, doing so will prevent us from properly representing UNITY inference rules in HOL. The reason is rather technical, see for example [2]. A standard way to get around this is to represent a state-predicate semantically in HOL as a function from some type `'s` representing the universe of states to the type `bool` [1, 18, 29, 2]. So, we have another set of operators: `AND`, `OR`, `IMP`, `NOT`, `!!`, and `??` which are just the previously listed boolean operators lifted to the state predicate level. For example, `AND` is defined as:

```
p AND q  =  (\s. p s /\ q s)
```

Abstractly though, the reader can pretend that both sets of operators are equivalent. If `p` is a predicate (over some type `'s`), `valid p` is defined as follows:

```
valid p = !s. p s
```

meaning that `p` is predicate that holds for all states in the assumed universe of states. Given some type `'s` representing the universe of states, an action is modelled as a relation over `'s`. So it is a function of type `'s->'s->bool`. Consequently, Hoare triples can be defined as follows:

4 Examples of UNITY variants

The definition of ∀UNITY is given in Section 5, in this section we first take a look at two concrete examples of UNITY variants which can (quite easily) be obtained from ∀UNITY . The first example is Sanders' UNITY [24] which extends classical UNITY with a new ability. The final example is a UNITY variant called COMUNITY, proposed by Prasetya *et al.* in [20], which boosts UNITY even further using a relatively simple extension.

4.1 GLEADSTO

When instantiating ∀UNITY, typically we define the `LEADSTO` operator in the same way as in classical UNITY, namely as the smallest transitive and left-disjunctive closure of the `ENSURES` relation. However, in a particular ∀UNITY instance, we may want to use a base relation that is different than the classical `ENSURES`. Given a binary relation `U`, in ∀UNITY the smallest transitive and left-disjunctive closure of `U` is written as `GLEADSTO U` –see Appendix A for a formal definition.

4.2 Sanders' UNITY

In classical UNITY we cannot use our implicit knowledge about the invariant of a program to simplify a given UNITY specification. In [24] Sanders offers a simple solution to this, she extends all UNITY operators with a new parameter that specifies an invariant and that subsequently can be used to simplify the other parameters.

Here a program `P` will be represented by a pair `(A,init)` where `A` represents the program's set of actions and `init` is a predicate specifying the program's initial condition. `ACTIONS P` and `INIT P` will return `A` and `init` respectively. The notion of an invariant is defined as follows (note that the definition below is weaker that the one originally used in [24] because the latter showed to be unsound [17]):

```
INV P J  =  valid((INIT P) IMP J)  /\  (!a. ACTIONS P a ==> HOA J a J)
```

Here is the concrete definition of Sanders' UNITY.

Definition 2. : Sanders' UNITY

1. `UNLESS P J p q =`
   `INV P J /\ (!a. ACTIONS P a ==> HOA (J AND p AND NOT q) a (p OR q))`
2. `ENSURES P J p q =`
   `UNLESS P J p q /\ (?a. ACTIONS P a ==> HOA (J AND p AND NOT q) a q)`
3. `LEADSTO P J = GLEADSTO (ENSURES P J)`

4.3 COMUNITY

COMUNITY (COMpositional UNITY) [20] increases the power of Sanders UNITY even further:

1. In COMUNITY, a program is just represented by a set of actions. Compared to Sanders' UNITY, COMUNITY only requires the `J` parameter to be stable rather than invariant. A predicate `J` is stable in a program `P` if `P` cannot destroy it:

Evidently (see the definition of invariant in Subsection 4.2), a stable predicate that also holds initially is an invariant. By weakening the requirement of invariance of `J` to stability of `J`, in COMUNITY it is possible to specify properties of a program which may only be reachable when the program is executed in parallel with other programs.

2. COMUNITY allows the user to specify the sensitivity of a program property, say `X`, to external interference. This is specified in an additional parameter `A` consisting of a set of predicates that are indestructible by the environment. Obviously, for a program `P` the property `X` is preserved when `P` is composed with an environment that maintains (each predicate in) `A`. COMUNITY also comes with a set of (new) proof rules that, in the less trivial case, can be used to compose a program with an environment which can only maintain `A` temporarily[5].

Definition 3. :

1.
```
UNLESS P J A p q =
   STABLE P J /\ A p /\ A q /\
   (!a. P a ==> HOA (J AND p AND NOT q) a (p OR q))
```
2.
```
ENSURES P J A p q =
   UNLESS P J A p q   /\  (?a. P a ==> HOA (J AND p AND NOT q) a q)
```
3.
```
LEADSTO P J A = GLEADSTO (ENSURES P J A)
```

---

5 ∀UNITY

∀UNITY is the general version of the classical UNITY. It is general because it does not impose any concrete interpretation of UNITY operators. Instead, it gives a set of quite weak primitive inference rules (primitive properties) that abstractly model the minimal requirements to obtain a UNITY-like logic. We have proven that when all the primitive properties of ∀UNITY are satisfied, then all standard UNITY inference rules are valid. An arbitrary UNITY variant can be created by providing a concrete definition of the relation 'unless' and 'ensures' and then showing that they satisfy ∀UNITY primitive rules. We will refer to this process as *instantiating* ∀UNITY, and the resulting concrete UNITY logic will also be referred to as a UNITY *variant* or an *instance* of ∀UNITY.

For each derived inference rule in ∀UNITY, we specify the minimal set of primitive properties that have to be satisfied in order for the inference rule to be valid. So, a user can create an instance that does not satisfy some primitive properties, or only conditionally satisfies them. In the first case, the created instance simply inherits less derived rules. In the second case, for example when a primitive property R only holds under a certain condition C, we can always propagate the condition C to all derived rules that depend on R.

Notice that both UNITY variants described in Section 4 specify a program property using an expression of the form `UOP V1 ... Vn p q` where: `UOP` is a UNITY operator (`UNLESS`, `ENSURES`, or `LEADSTO`); `V1` ... `Vn` are what we will refer to as *extensional parameters*; and `p` and `q`, from now on called the `pq`-*parameters*, are those parameters that all UNITY variants agree on (e.g. in all variants `LEADSTO ... p`

[5] For example, suppose `A` can be maintained by the environment `Q` during a time interval which is characterized by the state predicate `a`; suppose the program `P` can do the progress `LEADSTO P J A p q`. We can infer that the composition `P PAR Q` will satisfy:

q specifies a progress property in which if p holds then eventually q will also hold). For example, a classical UNITY property mainly depends on the pq-parameters, having only one extensional parameter that models the program[6] the property is related to. Other UNITY variants add new abilities by adding more extensional parameters and defining their relation with the pq-parameters. This is a very general form of UNITY properties, and all UNITY variants whose operators can be written in this form can be instantiated from ∀UNITY.

∀UNITY only focuses on the pq-parameters, because the interpretation of the extensional parameters and their relation with the pq-parameters are variant specific and hence beyond the scope of ∀UNITY. However, we cannot completely abstract away from the extensional parameters, since, whatever they are, the information in the pq-parameters is some way related to these extensional parameters. To capture this relation between the pq-parameters and the extensional parameters, ∀UNITY has a new operator called `implies`, which, as the name suggests, behaves in many ways like the ordinary ⇒ operator. The characterization of `implies` is given in the next Subsection 5.1. Furthermore, Subsection 5.6 provides a number of theorems capturing some general forms of inference rules concerning extensional parameters.

In the examples of UNITY variants in Section 4, the UNITY operators are concretely defined, that is `UNLESS`, `ENSURES` and `LEADSTO` are (HOL) constants. In ∀UNITY, however, they are not concretely defined, and in all inference rules of ∀UNITY they are (implicitly) universally quantified variables. As indicated before, to emphasis this distinction, in ∀UNITY the operators are written with lower case characters, e.g. `unless` and `ensures`. Note that these variables model binary operators: they only have the p and the q parameter.

5.1 The Implies Operator

As indicated above, ∀UNITY needs one more operator to be specified, namely `implies`. It is used to specify the fact that in some given temporal situation a state predicate p implies another state predicate q. This is mainly used to simplify a specification. For example, suppose a program has the property `leadsto p q`. If we also have `implies q r` and `implies r q` then we know that in 'this situation' q and r are equivalent, and hence q can be simplified to r. The situation under which the implication holds is typically specified in the extensional parameters. For example, in the Sanders' UNITY this is specified in the J parameter:

Definition 4. : Sanders' Implies

```
IMPLIES J p q  =  valid (J AND p IMP q)
```

In Sanders' UNITY J is intended to be an invariant of a program. The above concrete definition of `implies` means that we are allowed to use what we know about the program's invariants to infer the implication, and ultimately, to simplify the pq-parameters of the UNITY specifications of the program.

In the classical UNITY, we do not have the J parameter and `implies` corresponds to the usual predicate logic IMP:

Definition 5. : Classical Implies

```
IMPLIES p q  =  valid (p IMP q)
```

[6] The fact that the program is an extensional parameter, means that in ∀UNITY there is

COMUNITY has a more restricted notion of `implies`, which allows us to infer implication only when certain properties of the environment are also satisfied:

Definition 6. : COMUNITY Implies

```
IMPLIES J A p q  =  A p /\ A q /\ valid (J AND p IMP q)
```

5.2 Domain of the Operators

In the classical UNITY the `p` and `q` in, for example, `UNLESS p q` can be any predicate. So, the *domain* of the operator is simply the entire universe of predicates. However, in other UNITY variants this may not be the case. For example, above we have seen that in COMUNITY the domain of the `pq`-parameters of the `IMPLIES` operator is also restricted by `J` and `A`. Consequently, it is useful to introduce the notation `inDomain U` to denote the domain of a UNITY operator `U`. Although there are a number of ways to characterize `inDomain`, we choose the following. Observe that 'reflexiveness' is a desired (natural) property for any UNITY operator. That is, for any `p` in the domain of a given UNITY operator `U`, we want `U p p` to be a valid property –but only, for `p`'s which are in the domain of `U`. So we can just as well characterize `inDomain` as follows:

Definition 7. : In Domain

```
inDomain U p = U p p
```

In general, this is not a complete characterization of 'domain', however for UNITY operators it is.

5.3 ∀UNITY Primitive Rules

Below we list the primitive inference rules (properties) of ∀UNITY. The properties will be divided into three groups: the I, D, and N groups. Some informal explanation is provided. Their formal definition is listed in Appendix A.

I Properties Abstractly, the `implies` operator behaves just like the ordinary ⇒ operator. They are however not equivalent. In a given UNITY variant `implies` may use information in the extensional parameters to conclude the implication. The following properties specify how much of the ⇒-behavior we need in `implies`.

1. `isClosed_under_PredOPS (inDomain implies)`
   This says that `implies` should be closed under the standard predicate logic operators.
2. `includesIMP implies`
   This requirement says that if `p ==> q` holds, then `implies p q` should also

D Properties These are domain constraining properties. A typical domain requirement in ∀UNITY is the one that says, for example, that if `unless p q` holds then both `p` and `q` must be in the domain of `unless`. This is written `hasProperDomain unless` in ∀UNITY . Another example is the requirement that says that the domain of `unless` should be included in the domain of `implies`:

```
!p. inDomain unless p ==> inDomain implies p
```

Since `implies` provides the link to the extensional parameters, it is this kind of requirement that ensures that the `pq`-parameters are always consistent with the extensional parameters. So, if we recall the COMUNITY example from section 4.3, the above requirement ensures that in `UNLESS P J A p q`, both `p` and `q` are members of `A`. Below, we list the D properties:

1. `hasProperDomain implies`
2. `hasProperDomain unless`
3. `!p. inDomain unless p ==> inDomain implies p`
4. `hasProperDomain ensures`
5. `!p. inDomain ensures p ==> inDomain implies p`

N Properties In all three UNITY variants from the previous section, `unless` and `ensures` are defined in terms of the next-state behavior of the specified program. However, the next-state behavior of the program may depend on the situation derivable only from the information in the extensional parameters. This information is abstracted away in ∀UNITY, so obviously in ∀UNITY we cannot define the operators in the same way. Fortunately, there is another way. The following primitive properties abstractly characterize the intended temporal properties described by ∀UNITY operators:

1. `isSubRelationOf implies unless`
   This property indicates that for all `p` and `q`, if `implies p q` then `unless p q`.
2. `satisfiesUNLESS_AntiRefl unless`
   This states that `unless` is anti-reflexive. So, `unless p (NOT p)` is always a valid property, provided `p` is in the domain of `unless`.
3. `satisfiesUNLESS_Conj unless`
   This property expresses that `unless` satisfies the General Conjunction rule –see Theorem 1.
4. `satisfiesUNLESS_Disj unless`
   This says that `unless` satisfies the General Disjunction rule [4], which is the dual of the the General Conjunction rule.
5. `satisfiesUNLESS_Subst implies unless`
   This property reflects that `unless` satisfies the Substitution rule [4]. More precisely, if the following hold:
   `implies p q /\ implies q p /\ implies r s`
   Then we can replace `unless q r` with `unless p s`. Notice that the condition for substitution is expressed in terms of `implies` whose concrete definition is left unspecified in ∀UNITY . As remarked earlier, in Sanders' UNITY `implies` can be expected to carry information about a program's invariant. In the classical UNITY `implies` is simply `IMP`[7].
6. `isSubRelationOf implies ensures`
7. `satisfiesPSP ensures unless`
   This indicates that `ensures` also satisfies the Progress Safety Progress (PSP) rule [4]. This property is necessary for deriving the PSP rule for `leadsto`.

[7] So, even for the classical UNITY we get the Substitution rule from ∀UNITY , though

8. `isSubRelationOf ensures unless`
   This property reflects that `ensures` is a more restricted form of `unless`. Most UNITY variants take this property for granted: `ensures` is just `unless` strengthened with a requirement on the existence of some helpful action/transition. Curiously, the only derived inference rule that depends on this property is the Completion rule.
9. `leadsto = GLEADSTO ensures`
   This says that `leadsto` has to be defined as the least transitive and left-disjunctive closure of `ensures`.

5.4 Instantiating ∀UNITY

As an example, to instantiate ∀UNITY to COMUNITY, we substitute:

```
implies  with  IMPLIES P J A
unless   with  UNLESS P J A
ensures  with  ENSURES P J A
leadsto  with  LEADSTO P J A
```

where `IMPLIES` is defined in Definition 6; the other upper case operators are defined in Subsection 4.3. We should also substitute `GLEADSTO ensures` in ∀UNITY rules with `LEADSTO P J A`.

5.5 Derived Inference Rules

To give some idea, below we show some of the derived inference rules of ∀UNITY —the complete list of rules can be found in the distribution package that contains much more rules. The following is ∀UNITY 's version of the general conjunction rule for `unless`.

Theorem 3. : Unless Simple Conjunction Rule

```
1  |- includesIMP implies                          /\
2     isClosed_under_PredOPS (inDomain implies)   /\
3     (!p. inDomain unless p ==> inDomain implies p) /\
4     hasProperDomain unless                      /\
5     satisfiesUNLESS_Subst implies unless        /\
6     satisfiesUNLESS_Conj unless                 /\
7     unless p1 q1                                /\
8     unless p2 q2
9     ==>
10    unless (p1 AND p2) (q1 OR q2)
```

As another example, below is ∀UNITY's version of the PSP rule. Notice ∀UNITY primitive properties appearing as assumptions of both rules (the first six assumptions in ∀UNITY 's Unless Simple Conjunction Rule and the first eight assumptions in ∀UNITY 's PSP Rule). These are the primitive properties required to obtain the standard form of those rules (as in Theorems 1 and 2). So, when instantiating ∀UNITY, these primitive properties will have to be discharged, to produce the standard, familiar form of those rules.

```
|- includesIMP implies              /\
   isClosed_under_PredOPS (inDomain implies)   /\
   hasProperDomain unless           /\
   (!p. inDomain unless p ==> inDomain implies p) /\
   hasProperDomain ensures          /\
   isSubRelationOf implies ensures      /\
   (!p. inDomain ensures p ==> inDomain implies p) /\
   satisfiesPSP ensures unless          /\
   GLEADSTO ensures p q             /\
   unless a b
   ==>
   GLEADSTO ensures (p AND a) (q AND a OR b)
```

5.6 Inferring and Composing Extensional Parameters

As mentioned earlier in this section, ∀UNITY's focus is only on the `pq`-parameters. Consider a COMUNITY specification `LEADSTO P J A p q`. Recall that the `p` and `q` are called the `pq`-parameters and that the others (e.g. `P`, `J` and `A`) are called extensional parameters. The `implies` operator can be used to capture the relation between the two kind of parameters, but beyond that ∀UNITY basically does not provide any support to reason about the extensional parameters since that kind of reasoning is variant specific. However, there are two situations in which it is desirable to be able to interfere with the extensional parameters. We will illustrate these with COMUNITY examples:

1. Some program properties implicitly imply properties of the extensional parameters. For example, `LEADSTO P J A p q` implies that `J` is stable in `P`.
2. Sometimes the value of an extensional parameter can be changed without destroying the validity of the underlying property. For example, `LEADSTO P J A p q` is preserved if we strengthen `J` with another stable predicate. The property is also preserved if we compose `P` and `A` with another program and another set of predicates provided that these satisfy certain constraints [20]. Inference rules for program composition, e.g. the Union rules [4] and Singh rules [25, 13, 20], fall into this category.

∀UNITY provides two theorems to accommodate these inferences with the extensional parameters. Evidently, these theorems do not characterize the specific conditions under which the interferences are applicable since this is something which is variant specific. Before we can state the theorems we first need to introduce the following two operators:

Definition 8. : Implicitly Implies

```
implicitlyImplies unityOp property  =  !A p q. unityOp A p q ==> property A
```

Definition 9. : Conservative Extension

```
isConservative unityOp extend
=
!A p q. unityOp A p q ==> unityOp (extend A) p q
```

So, for example: `implicitlyImplies (\J LEADSTO P J A) (STABLE P)` captures what we said earlier, namely that in COMUNITY `LEADSTO P J A p q` implies the stability of `J P`; and `isConservative (\J. LEADSTO P J A) (\J. J AND J')` states that in COMUNITY the property `LEADSTO P J A p q` is preserved when we strengthen `J` to `J AND J'` (for some fix `J'`). It is then quite trivial to obtain the following theorems:

Theorem 5. : Implicitly Implies Rule

```
|- implicitlyImplies unityOp property  /\  unityOp A p q
    ==>
   property A
```

Theorem 6. : Conservative Extension Rule

```
|- isConservative unityOp extend  /\  unityOp A p q
   ==>
   unityOp (extend A) p q
```

When `unityOp` is a leads-to operator, we can even derive two stronger theorems. The first theorem states that if a property is preserved by a given `ensures` relation, it will also be preserved by the corresponding `leadsto` relation. The second indicates that if `A` defines a way to conservatively extend a given `ensures` relation, it will also be a conservative extension to the corresponding `leadsto` relation.

Theorem 7. :

```
|- implicitlyImplies ensures property  /\  GLEADSTO (ensures A) p q
    ==>
   property A
```

Theorem 8. :

```
|- isConservative ensures extend  /\  GLEADSTO (ensures A) p q
   ==>
   GLEADSTO (ensures (extend A)) p q
```

---

6 Conclusion

∀UNITY is a generalization of UNITY. It provides the same set of inference rules as standard UNITY but it does not impose any concrete interpretation of `unless` and `ensures`. Instead, it gives a set of weak primitive properties that abstractly model the minimalistic requirements to obtain a UNITY-like logic. A user can create an arbitrary variant of UNITY by providing a concrete definition of `unless` and `ensures` and then showing that they satisfy ∀UNITY primitive rules.

Furthermore, for each derived inference rule in ∀UNITY, we specify which primitive properties it minimally requires. Hence, it is possible to create a weaker ∀UNITY instance where not all primitive properties are satisfied, or only conditionally satisfied.

∀UNITY is suitable for creating a UNITY variant whose properties are specified like: `UnityOperator V p q` where `p` and `q` are state predicates having the usual UNITY meaning and `V` is a list of additional parameters. This is a very general form

UNITY variants are constructed in this way, e.g. Collete and Knapp's variants [7, 8], closures operators in the new UNITY [14], and COMUNITY [20].

∀UNITY is also useful for constructing general theories: an application theory which is based purely on ∀UNITY will also be valid for all UNITY variants which can be instantiated from ∀UNITY .

Summing up, creating a UNITY variant using ∀UNITY the user can benefit from the following advantages:

1. The availability of a rich set of standard inference rules that have already been proved, saving the user a lot of work.
2. The security of the fact that all the inference rules have been proved *mechanically* in HOL, making it very safe to use them.
3. The convenience of the presentation as a HOL library, providing the user automatically with all theorem proving support of HOL.

A Definition of ∀UNITY Primitive Properties

```
1. |- isClosed_under_PredOPS dom
       =
       dom TT  /\  (!p. dom p ==> dom (NOT p))  /\
       (!p q. dom p /\ dom q ==> dom (p AND q)) /\
       (!W. (!p. W p ==> dom p) ==> dom (!!p::W. p))
2. |- includesIMP U
       =
       !p q. inDomain U p /\ inDomain U q /\ valid (p IMP q) ==> U p q
3. |- hasProperDomain U
       =
       !p q. U p q ==> inDomain U p /\ inDomain U q
4. |- inDomain U p = U p p
5. |- isSubRelationOf U V = !x y. U x y ==> V x y
6. |- satisfiesUNLESS_AntiRefl unless
       =
       !p. inDomain unless p ==> unless p (NOT p)
7. |- satisfiesUNLESS_Conj unless
       =
       !p1 q1 p2 q2.
        unless p1 q1 /\ unless p2 q2
        ==>
        unless (p1 AND p2) (q1 AND p2 OR q2 AND p1 OR q1 AND q2)
8. |- satisfiesUNLESS_Disj unless
       =
       !p1 q1 p2 q2.
        unless p1 q1 /\ unless p2 q2
        ==>
        unless (p1 OR p2) (q1 AND NOT p2 OR q2 AND NOT p1 OR q1 AND q2)
9. |- satisfiesUNLESS_Subst implies unless
       =
       !p q a b.
        unless p q /\ implies p a /\ implies a p /\ implies q b
        ==>
        unless a b
10. |- satisfiesPSP progress unless
       =
       !p q a b.
```

```
11. |- GLEADSTO ensures p q
       =
       !U. isSubRelationOf ensures U /\  isTransitive U /\ isLeftDisj U
           ==>
           U p q
12. |- isLeftDisj leadsto
        =
        !W q. (?p. W p)  /\  (!p. W p ==> leadsto p q)
              ==>
              leadsto (??p::W. p) q
```

B Relation with New UNITY

In [14], Misra introduces New UNITY which is based on a set of new temporal operators: **co**, **transient**, **en**, and **leadsto**. The **leadsto** and **en** behave the same way as their classical counterparts, and **transient** is just an auxiliary operator used to define **en**. The operator **co** is an alternative to **unless**. Although, **co** has nicer algebraic properties, **unless** has a more intuistic and familiar interpretation. The choice between them is probably a matter of taste. They can be defined in terms of each other. For example, here is how **co** can be defined in terms of **unless** in ∀UNITY :

Definition 10. : CO

```
|- !P J A p q.
    CO implies unless p q = implies p q /\ unless p (NOT p AND q)
```

□

New UNITY also introduces the notion of *closure* which provides an abstraction for program composition. For example, the closure of the **leadsto** operator is called **cleadsto**, defined as follows:

$$P \vdash p \textbf{ cleadsto } q \;=\; (\forall Q :: P[\!]Q \vdash p \textbf{ leadsto } q)$$

Unlike in classical UNITY, in New UNITY $P[\!]Q$ is only defined if Q satisfies P's link constraint. The link constraint of a program essentially specifies some upper bound on the kind of operations the environment can do on the variables of P.

New UNITY's closured operators can be expressed in terms of COMUNITY operators. So, $P \vdash p$ **cleadsto** q can be written as `LEADSTO P TT A p q` where `A` is suitably chosen to represent P's link constraint. Since COMUNITY is an instance of ∀UNITY –it satisfies all ∀UNITY primitive rules– so is the closured logic of New UNITY's closured operators.

References

1. F. Andersen. *A Theorem Prover for UNITY in Higher Order Logic.* PhD thesis, Technical University of Denmark, 1992.
2. A. Azurat and I.S.W.B. Prasetya. A survey on embedding programming logics in a theorem prover. Technical Report UU-CS-2002-007, Inst. of Information and Comp. Science, Utrecht Univ., 2002. Download: `www.cs.uu.nl/staff/wishnu.html`.
3. I. Chakravarty, M. Kleyn, T.Y.C. Woo, R. Bagrodia, and V. Austel. UNITY to UC: A case study in the derivation of parallel programs. *Lecture Notes of Computer Science*, 574:7–20, 1991.
4. K.M. Chandy and J. Misra. *Parallel Program Design – A Foundation.* Addison-Wesley

5. C. Christian and R. Gruia-Catalin. Formal specification and design of a message router. *ACM Transactions on Software Engineering and Methodology*, 3(4):271–307, October 1994.
6. P. Collette. Composition of assumption-commitment specifications in a UNITY style. *Science of Computer Programming*, 23:107–125, December 1994.
7. P. Collette. Composition of assumption-commitment specifications in a UNITY style. *Science of Computer Programming*, 23(2–3):107–125, December 1994. Selected papers of the Colloquium on Formal Approaches of Software Engineering (Orsay, 1993).
8. P. Collette and E. Knapp. Logical foundations for compositional verification and development of concurrent programs in UNITY. *Lecture Notes in Computer Science*, 936:353–??, 1995.
9. Mike J.C. Gordon and Tom F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.
10. R. Gruia-Catalin and P. J. McCann. An introduction to mobile UNITY. *Lecture Notes in Computer Science*, 1388:871–880, 1998.
11. P.J.A. Lentfert and S.D. Swierstra. Distributed maximum maintenance on hierarchically divided graphs. *Formal Aspects of Computing*, 5(1):21–60, 1993.
12. Zhiming Liu. Modelling checkpointing and recovery within UNITY. Research Report CS-RR-145, Department of Computer Science, University of Warwick, Coventry, UK, August 1989.
13. J. Misra. Notes on UNITY 17-90: Preserving progress under program composition. Downloadable from `www.cs.utexas.edu/users/psp`, July 1990.
14. J. Misra. *A Discipline of Multiprogramming*. Springer-Verlag, 2001.
15. A. Pizzarello. An indrustial experience in the use of UNITY. *Lecture Notes of Computer Science*, 574:38–49, 1991.
16. I. S. W. B. Prasetya. *Mechanically Supported Design of Self-stabilizing Algorithms*. PhD thesis, Inst. of Information and Comp. Science, Utrecht Univ., 1995. Download: `www.cs.uu.nl/library/docs/theses.html`.
17. I.S.W.B. Prasetya. Error in the UNITY substitution rule for subscripted operators. *Formal Aspects of Computing*, 6:466–470, 1994.
18. I.S.W.B. Prasetya. Formalizing UNITY with HOL. Technical Report UU-CS-1996-01, Inst. of Information and Comp. Science, Utrecht Univ., 1996. Download: `www.cs.uu.nl/staff/wishnu.html`.
19. I.S.W.B. Prasetya. Mechanically verified self-stabilizing hierarchical algorithms. *Lecture Notes in Computer Science*, 1217:399 – 415, 1997.
20. I.S.W.B. Prasetya, T.E.J. Vos, S.D. Swierstra, and B. Widjaja. A theory for composing distributed components based on mutual exclusion, 2002. Submitted for publication. Download: `www.cs.uu.nl/~wishnu`.
21. J.R. Rao. *Extensions of the UNITY methodology: compositionality, fairness, and probability in parallelism*, volume 908 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1995.
22. D.S. Richardson and M. Abrams. UNITY algorithms for detecting stable and nonstable terminationconditions in time warp parallel simulations. Technical Report TR-90-62, Virginia Polytechnic Inst. and State University, 1990.
23. B.A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3(2):189–205, 1991.
24. B.A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3(2):189–205, 1991.
25. A.K. Singh. Leads-to and program union. *Notes on UNITY*, 06-89, 1989.
26. M.G. Staskaukas. The formal specification and design of a distributed electronic funds-transfer system. *IEEE Trans. on Computers*, 37(12):1515–1528, December 1988.
27. R.T. Udink. *Program Refinement in UNITY-like Environments*. PhD thesis, Inst. of Information and Computer Sci., Utrecht University, 1995. Downloadable from `www.cs.uu.nl`.
28. R.T. Udink and J. N. Kok. The RPC-Memory specification problem: UNITY + refinement calculus. *Lecture Notes in Computer Science*, 1169:521–??, 1996.
29. T.E.J. Vos. *UNITY in Diversity: A Stratified Approach to the Verification of Distributed Algorithms*. PhD thesis, Inst. of Information and Computer Sci., Utrecht

30. T.E.J. Vos and S.D. Swierstra. Proving distributed hylomorphisms. Technical Report UU-CS-2001-40, Inst. of Information and Comp. Science, Utrecht University, 2001. Download: `www.cs.uu.nl`.
31. T.E.J. Vos, S.D. Swierstra, and I.S.W.B. Prasetya. Formal methods and mechanical verification applied to the development of a convergent distributed sorting program. Technical Report UU-CS-1996-37, Inst. of Information and Computer Sci., Utrecht University, 1996. downloadable from `www.cs.uu.nl`.

Part III

Integrating Model Checking

Verification of Statecharts Including Data Spaces

Steffen Helke and Florian Kammüller

Technische Universität Berlin
Institut für Softwaretechnik und Theoretische Informatik

Abstract. Hierarchical Automata represent a structured model of statecharts. They are already formalized in Isabelle/HOL. The present work contributes to this formalization in three respects. First, it integrates data spaces into the formalization and resolves racing effects using a novel method. Second, connecting Isabelle/HOL and the model checker SMV, it proposes a more pragmatic way to obtain an efficient representation of finite data spaces in the input language of SMV. Finally we show how to apply a previously proposed algorithm for the generation of property-preserving data abstractions.

1 Introduction

One attempt on supporting the mechanized verification of statecharts in recent years was to connect commercial simulation tools for statecharts [HN96] to model checkers [MLS97,Hie98,BW98,BG98]. Model checking techniques are inherently restricted to finite state spaces. However statecharts defined on data variables, for instance integer variables, represent infinite state systems. To avoid this contradiction the known approaches simply restrict statecharts by omitting data spaces. This restriction is hard, because case studies containing data are the norm rather than the exception in realistic applications.

Our approach incorporates data spaces into an Isabelle/HOL mechanization of statecharts [HK01]. Like [MLS97] it is based on Hierarchical Automata (HA), a representation of statecharts that makes the hierarchy of states more explicit and thereby provides an elegant way of resolving interlevel transitions. Those transitions between states on different levels of hierarchy pose difficult problems in classical statecharts [HN96], when calculating priorities.

The incorporation of data spaces into the Isabelle/HOL formalization of HA is in principle straightforward. However, in order to deal with racing problems, special intermediate data values have to be introduced into the update functions to resolve conflicting changes to data variables. This approach to handle the well-known racing problem can be dealt with rather elegantly in the logic. Still, as statecharts may be fairly complex it is cumbersome to prove properties of concrete applications as this boils down to proving properties in the semantics given by a step relation over the status. Here, we suggest the additional use of model checkers and devise an extension of the existing representation of HA for SMV extending it by data as well.

In this paper we first illustrate the racing problem on a simple example in Section 2. After briefly summarizing the existing formalization of HA in Section 3, we introduce in detail the formalization of HA with data spaces in particular presenting how racing is resolved. To show that the extension by data does not restrict the ability to model check we outline in Section 4 the representation for the SMV model checker that may be derived from the Isabelle/HOL formalization. Finally in Section 7, we sketch briefly how the derivation of the model checkable version of the HA can be derived schematically using well known abstraction techniques pointing out limitations and describing possible solutions for the particular application to statecharts with data spaces. Section 8 contains related work and conclusions.

2 Example Specification: Racing Effects

The following example of a statechart S_1 composed of the three parallel subsystems S_2, S_3 and S_4 illustrates the racing effect. We consider the semantic continuations, in which the states $S_1, S_2, S_3, S_4, S_5, S_7$ and S_9 are initially activated and the events e_1 and e_2 are enabled. In this situation three transitions fire (as all three have one of the events e_1 and e_2 as preconditions) and the target states S_6, S_8 and S_{10} are reached simultaneously. One possible outcome of the parallel execution of the action parts of the transitions (the statements and events after the / in the attached transition labels) is that the data variable y is written concurrently by the update functions $update_1$ and $update_3$. This effect is called *racing* in the literature.

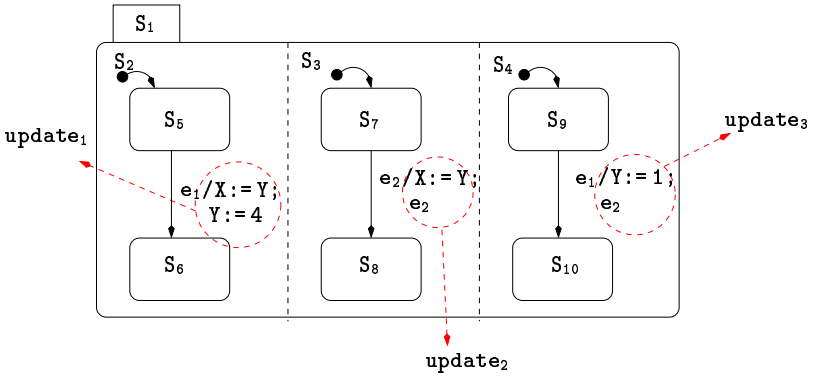


Fig. 1. Statecharts Specification with Racing Effects

Figure 2 illustrates for this example how we can solve such a conflict by an interleaving semantics. We consider all next data statuses that are possible outcomes of the concurrent writing process. The assignment X:=Y overwrites the

old value of X_{pre} with Y_{pre} in both cases, but the second component is once set to 4 and once to 1.

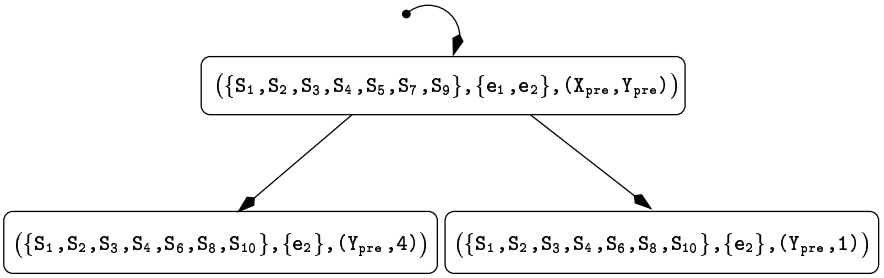


Fig. 2. Semantic Interpretation for Solving Racing Effects

The quest for a formalization of statecharts with data spaces is to model the interleaving that is diagrammatically easily explained equally simply in the logical representation of statecharts. The problem in formalizing is that update functions are usually partial functions, i.e. they only change certain components. For a concrete statechart this problem has a straightforward solution because the concrete components are known. However, for a general polymorphic definition of statecharts these components cannot be identified while we still need to totalize them. The basic idea of our solution is the use of intermediate values as parameters of update functions. Before we introduce this solution we give a short summary of the existing formalization that lays the foundations.

3 Representing Hierarchical Automata in Isabelle/HOL

Hierarchical Automata represent a structured model of statecharts. They are already formalized in Isabelle/HOL [HK01]. This work is restricted to event driven systems. First we give in this section a brief summary of the original formalization of Hierarchical Automata (HA). In Section 4 we introduce the extension by data spaces in full detail.

The basic building block for HA are sequential automata, a simple unnested form of state automata just comprising a set of states, an initial state, and a transition relation with labels. The labels correspond to statecharts labels, *i.e.* contain a proposition — describing preconditions over events and data for a transition — and an action part describing the effects of this transition. Sequential automata (SA) are composed into a hierarchical automaton using a composition function γ. The composition is defined as a function from states to sets of SA. The SA contained in $\gamma(s)$ for some state s of a HA are supposed to be parallel automata refining the state s. The composition function represents a tree-like structure: as a root it has a single sequential automaton, called the

root automaton γ_{root}, the sets of states of the composed SA are pairwise disjoint, each SA different from root composed into the HA by γ has exactly one ancestor, and γ contains no cycles.

For an intuitive illustration of sequential automata and their composition into an HA, consider Figure 3. The arrows represent the composition function γ. The components Root_CTRL, S_2_CTRL, S_3_CTRL, and S_4_CTRL are the constituting sequential automata where the latter three are parallel subcomponents of Root_CTRL. A part from the composition function γ a HA is defined by its constituting set of SAs and an event set. The formal definitions of SA and HA will be considered in more detail, and with the extension of data spaces, in the next section.

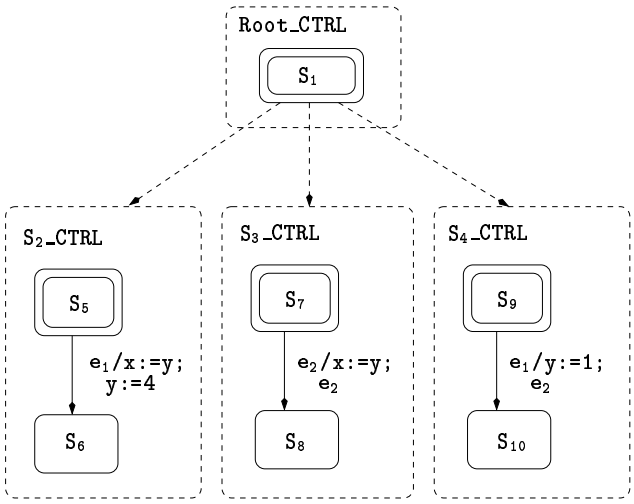


Fig. 3. Hierarchical Automata with Racing Effects

In [HK01] we further refine the definition of HA into one using the Isabelle/HOL datatype package introducing a tree datatype replacing the mathematically elegant definition of γ. This second version of HA is more efficient than the function based structure representation of HA.

Concerning the well-formedness of HA we introduce another method used to make reasoning more efficient. We define constructors that enable the construction of HA step by step from existing well-formed HAs by adding single SAs. Besides providing a structured approach to constructing HAs this method also tackles the rather complex well-formedness conditions. We define these constructors for the function-based version and for the tree-based representation of HA.

Finally, we define a step semantics for HA in Isabelle/HOL based on the notion of status of a HA. A status is built by a configuration — the set of current

states — plus the HA itself and a set of current events. The configurations are again defined based on a tree datatype thereby providing good support for reasoning by defining primitive recursive evaluation functions for the step relation on statuses. The semantical representation of configurations is wrapped up using inductive sets to define reachable states of the HA.

The advantages gained by this formalization are that we now have an adequate model of statecharts that has been designed as a common input language for model checkers, and we can use this model in Isabelle for reasoning about higher order properties. The theorem proving is made easier by providing the more efficient representation using the tree datatype. At the same time, when model checking the same statechart we know that it is consistent with a logical representation. To support consistency we preserve both versions of the formalization and relate them with HOL functions enabling the translation of properties.

In the following, we will for reasons of clarity introduce the novelties concerning data spaces on the less efficient but mathematically more concise version of the HA formalization. However, it should be noted that we have defined the current extensions for both representations.

4 Extending the Formalization by Data Spaces

Sequential automata (SA) are again the building block for hierarchical automata used in the current extension by data. Compared to the previous formalization [HK01], data are simply introduced as a further polymorphic component δ of the types $(\sigma,\varepsilon,\delta)$ `seqauto` of sequential automata and $(\sigma,\varepsilon,\delta)$ `hierauto` of hierarchical automata. The type of HA is further extended by an additional component representing an initial data value as is usual for statecharts. More decisive changes are introduced for the labels, most prominently the extension of the action part leads to the definition of update functions as part of the labels (see below).

According to the original formalization of HAs we have also adapted the construction operators. Assuming that for the HA of the running example an initial data value `(8,7)` and the SAs `Root_CTRL`, $\mathtt{S_2\_CTRL}$ and $\mathtt{S_3\_CTRL}$ are given, we can define the data containing HA in Isabelle/HOL by construction operators as follows.

```
HA ::  (string, string, int * int) hierauto
HA ≡df (BasicHA Root_CTRL (8,7))
                  [+] (''S1'',S2_CTRL)
                  [+] (''S1'',S3_CTRL)
                  [+] (''S1'',S4_CTRL)
```

Note that the generic data type δ is instantiated by the cartesian product over integers to declare the data variables `X` and `Y` of our running example. In a first step the operator `BasicHA` builds a simple HA without hierarchy from the SA `Root_CTRL` and from the initial value. Refining the state $\mathtt{S_1}$ of `Root_CTRL` by the

subautomata S_2_CTRL and S_3_CTRL using the operator [+] we introduce step by step the hierarchy in the HA.

The racing problem is a semantical one. Therefore, the major adaptations have to be done to the formalization of the semantical model of HA, which is described by configurations and statuses. According to the extension of the type definition of HA by an initial data value, we add a data value to a status representing the current data assignment. This data component initially holds the initial value and is changed during the lifetime of the statecharts. We can now define the following type `status` for the semantic interpretation of HA, where the predicate `Status` describes that the configuration `C` respects the tree-like structure of the `HA` and the events in `E` are compatible to the events of the `HA`.

```
(σ,ε,δ) status ≡df {(HA,D,C,E) | (HA::((σ,ε,δ) hierauto))
                                 (D::δ)
                                 (C::(σ set))
                                 (E::(ε set)). Status HA D C E }
```

Based on this semantic status we need to define a set of traces that reflects the solution to the racing effects. Hence, this set of traces has to include precisely all those paths that are possible in the interleaving semantics. To realize this idea on a general level we have to deal with the generic data type and partial update functions. We assume a general type of data contained in a statechart and totalize the partial update functions using an additional auxiliary parameter which reflects results of concurring update functions. First, we represent the update functions on data variables by functions from an initial data value and from an auxiliary data value to the next value.

```
δ update ≡syn [δ,δ] => δ
```

Figure 4 shows the encoding of concrete update functions for the running example. The left hand side presents a definition of the update function $\texttt{update}_1$, in

```
update1 Daux Dpre ≡df          update2 Daux Dpre ≡df          update3 Daux Dpre ≡df
    let                              let                            let
      (Xaux , Yaux) = Daux             ...                            ...
      (Xpre , Ypre) = Dpre ;           ...                            ...
    in                               in                             in
      (Ypre , 4)                       (Ypre , Yaux)                  (Xaux , 1)
```

Fig. 4. Encoding of Update Functions for the Example of Figure 1

which the data variable `X` is updated by the auxiliary variable $\texttt{X}_{\texttt{aux}}$. In fact the data variable `X` should not be written, because $\texttt{update}_1$ is a partial function and

defines only an assignment for Y. The purpose of the auxiliary variables is to pass on an assignment of X, that was caused by concurrent update functions. In our example it could be the assignment of $update_2$. If there is no other update function that wants to write X, the variables X_{aux} and X_{pre} will be identified. In contrast to the partial function $update_1$, the function $update_2$ is total, which is reflected by the fact that no assignments by auxiliary variables are defined.

To integrate this idea in the semantic definitions of the original formalization of HAs, we manipulate the maximal non-conflicting sets of transitions. Those sets can be calculated for a current status according to well known transition selection algorithms [HN96]. Omitting data aspects, the transitions of those sets are not in conflict. However, transitions that write on data variables simultaneously can produce racing effects, which can cause additional conflicts. The following definition ResolveRacing solves racing conflicts for a current status ST and a given set of transitions TS by an interleaving semantics, using the special encoding of update functions.

```
ResolveRacing ST TS  ≡df
      let
         D = Value ST;
         FS = (λ F. F D) ‘ (Update TS);
         FSo = { f. (FS,f) : (foldSet ∘ id) };
         FSconst  = (λ F. (λ Daux Dpre. F D)) ‘ FSo;
         UpdateOverride = (λ F T.
                                 let
                                    (S1,L,S2) = T;
                                    (E,G,A)   = L
                                 in
                                    (S1,(E,G,(fst A, F)),S2))
      in
         (λ F. (UpdateOverride F) ‘ TS) ‘ FSconst
```

First, we determine the value of the current status. Second, we apply this current data value to all update functions, instantiating the second argument D_{pre}. The operator ‘ is defined in Isabelle as a map on sets. Considering our running example and starting from the current status $(\{S_1,S_2,S_3,S_4,S_5,S_7,S_9\},\{e_1,e_2\},(7,8))$ we obtain for FS the set $\{update_1\,(7,8),\ update_2\,(7,8),\ update_3\,(7,8)\}$. Third, we build all possible concatenations of update functions using the concatenation ∘ for functions. Note that the standard Isabelle distribution contains the operator foldSet that allows to solve this task in an easy way. The recursive anchor id describes the identity function. According to the running example we obtain the following set FS_o.

```
FSo = { update1 (7,8) ∘ update2 (7,8) ∘ update3 (7,8) ∘ id,
        update1 (7,8) ∘ update3 (7,8) ∘ update2 (7,8) ∘ id,
        update2 (7,8) ∘ update1 (7,8) ∘ update3 (7,8) ∘ id,
        update2 (7,8) ∘ update3 (7,8) ∘ update1 (7,8) ∘ id,
        update3 (7,8) ∘ update1 (7,8) ∘ update2 (7,8) ∘ id,
        update3 (7,8) ∘ update2 (7,8) ∘ update1 (7,8) ∘ id }
```

Fourth, we apply the current data value to each of the composed functions of $\mathtt{FS_O}$ to calculate constants that will again masquerade as update functions. Some functions of $\mathtt{FS_O}$ will be coincided in $\mathtt{FS_{const}}$, because the calculated constant will often be the same. For our running example we obtain the following two constant update functions.

$$\mathtt{FS_{const}} \equiv_{df} \{\ (\lambda\ D_{aux}\ D_{pre}.\ (8,4)),\ (\lambda\ D_{aux}\ D_{pre}.\ (8,1))\ \}$$

Finally, we define a function `UpdateOverride` to replace an original update function of a transition by a calculated constant update function. Using this function we generate a set of transition sets, that reflects the nondeterminism according to the interleaving semantics. For each entry in $\mathtt{FS_{const}}$ we obtain a duplicated transition set `TS`, in which all update functions are replaced by the corresponding constant update function. In our example, `ResolveRacing` results in the following set of transition sets.

$$\begin{array}{l}\{\ \{\ (S_4,\ (\{e_1\},\mathtt{true},(\{\},\ \ (\lambda\ D_{aux}\ D_{pre}.\ (8,4)))),S_5),\\ \quad\ (S_6,\ (\{e_1\},\mathtt{true},(\{e_2\},(\lambda\ D_{aux}\ D_{pre}.\ (8,4)))),S_7),\\ \quad\ (S_6,\ (\{e_2\},\mathtt{true},(\{e_2\},(\lambda\ D_{aux}\ D_{pre}.\ (8,4)))),S_7)\ \},\\ \ \ \{\ (S_4,\ (\{e_1\},\mathtt{true},(\{\},\ \ (\lambda\ D_{aux}\ D_{pre}.\ (8,1)))),S_5),\\ \quad\ (S_6,\ (\{e_1\},\mathtt{true},(\{e_2\},(\lambda\ D_{aux}\ D_{pre}.\ (8,1)))),S_7),\\ \quad\ (S_6,\ (\{e_2\},\mathtt{true},(\{e_2\},(\lambda\ D_{aux}\ D_{pre}.\ (8,1)))),S_7)\ \}\ \}\end{array}$$

Note that this approach to solve racing conflicts is elegant in HOL. However, an efficient representation of concrete statecharts for a model checker must be realized in a different way, which we present in section 6.

5 Representing Hierarchical Automata in SMV

In this section we sketch how hierarchical automata can be represented in the input language of SMV ("*Symbolic Model Verifier*") [McM93]. SMV is quite well suited for the description of state-based dynamic systems as it features a simple input language to define models as Kripke-structures. This definition comprises basically the declaration of state variables using `VAR` statements and the definition of their next states using `ASSIGNS` or `TRANS` statements. Additionally simple predicates may be defined by `DEFINE` to model the conditions on the control flow.

The encoding of HA presented in this section is very close to the procedure of Mikk [Mikk00]. In the next section we extend this approach by dealing with data variables and the resulting possibility of racing effects.

We illustrate the transformation steps according to our running example. First, we introduce for all sequential automata a so called automaton variable, which is declared over an enumeration type of all possible states in which the automaton can be. For our example we obtain three variables. Additionally we declare for each event a boolean variable, that reflects whether the event is enabled or not. Second, we define predicates for all state variables, that depend on each other to simulate the tree-like structure of configurations.

```
VAR   Root_CTRL  : {S1};          DEFINE  In_S1 := Root_CTRL = S1;
      S2_CTRL    : {S5,S6};              In_S5 := In_S1 & S2_CTRL = S5;
      S3_CTRL    : {S7,S8};              In_S6 := In_S1 & S2_CTRL = S6;
      S4_CTRL    : {S9,S10};             In_S7 := In_S1 & S3_CTRL = S7;
      En_e1      : boolean;              In_S8 := In_S1 & S3_CTRL = S8;
      En_e2      : boolean;              In_S9 := ...; In_S10 := ...;
```

Third, in order to reflect effects caused by nondeterminism adequately, we encode transitions by the TRANS-statement of the SMV input language. Where the use of the ASSIGNS-statement forces to describe the next state of a variable deterministically, the TRANS-statement enables the use of propositional formulas and equations to describe the next step in a more abstract way by imposing conditions on the next state. In the following we present the TRANS-rule encoding the transitions of the sequential automaton S_3_CTRL.

```
TRANS     S3_CTRL_active -->
             case
               In_S7:
                 case
                   S7_outgoing: (En_e2 & next(S3_CTRL) = S8);
                   1          :  next(S3_CTRL) = S7;
                 esac;
               In_S8: next(S3_CTRL) = S8;
             esac;
```

The predicate S_7_outgoing describes, whether the state S_7 can be exited or not.

```
DEFINE    S7_outgoing := In_S7 & En_e2
```

The variable S_3_CTRL_active is introduced to implement a selection algorithm of statecharts, which is usually based on an order of priorities for transitions. We will not consider this here in detail because it is not relevant for dealing with data variables.
To generate or dispose events Mikk defines predicates, which indicate, whether a transition is taken or not.

```
ASSIGNS   next(En_e2) := case
                           (T2_indicator | T3_indicator): 1;
                           1                            : {0,1};
                         esac
```

In our example an event e_2 is generated, if the transitions T_2 or T_3 are executed. This is defined in the condition part of the case statement by a disjunction. If the predicates of the transitions T_2 and T_3 are not satisfied we choose for the next step nondeterministically, whether the event e_2 is enabled or not. This treatment constitutes an open system semantics of statecharts, which demands that in each atomic time step the system can be triggered by external events generated by the environment. A closed system semantics would demand that in the second part of the case statement the event must be disposed of.

6 Implementing Data Spaces as Input for SMV

In this section we propose an extension of the representation of hierarchical automata in SMV by data spaces. As we have done it in Section 4 for the HOL representation we solve the racing problem that occurs when data variables are written simultaneously by transitions.

Model checking is a verification technique that is restricted to finite state spaces. Statecharts including data spaces represent usually an infinite state system because data variables are often defined on infinite data domains. To represent statecharts as hierarchical automata in SMV we must assume that the data domains are finite. Discussing data abstraction techniques in Section 7 we point out that this assumption is not unrealistic.

In the sequel we describe the procedure for implementing data spaces in SMV. First, we introduce for each component in the data space a data variable declared on a finite data domain. Second, we assign initial values to these variables.

```
VAR    X:  0 .. 255          ASSIGN   init(X) := 8;
       Y:  0 .. 255                   init(Y) := 7;
```

To solve racing effects, again picking up the idea of an interleaving semantics, we define updates on data variables by the TRANS-statement of the SMV input language using the indicator predicates introduced in Section 5 as follows.

```
TRANS  (T1_indicator & next(Y) = 4) | (T3_indicator & next(Y) = 1) |
       (! T1_indicator & ! T3_indicator & next(Y) = Y)
```

The TRANS-statement enables the description of the various alternatives that may result from the different transitions: if the first transition fires the value of Y is 4, if the third fires it is 1, otherwise it stays the same.

A schematic way of producing such TRANS-statements for the resolution of racing conditions has to process a HA systematically. Scanning a given hierarchical automaton, we collect the update functions containing a certain data variable. We connect all postconditions of the collected update functions by a disjunction. To ensure that a transition is really taken, we additionally demand for each update function the corresponding indicator predicate. This pragmatic procedure is leading to an efficient representation of data spaces in SMV avoiding additional technical overhead. For comparison, Büssow et al. [BG97] handle racing and persistency on data variables by variable locks and places. Places give identities to concurrent activities and locks define that a place has a write access. This approach is quite technical in contrast to our simple yet efficient solution.

7 Data Abstraction Techniques

Although the SMV representation of an HA illustrated by means of an example in the previous section can be derived schematically from the Isabelle/HOL

representation, it is a hand-made abstraction. It just overcomes the problem of infinite states by introducing finite domains for integer variables. In the present section we want to describe briefly how the abstraction process concerning infinite data-spaces can be performed using abstraction techniques and point out some ideas on how to guide the abstraction process and preserve the racing conditions.

Data abstraction is a technique based on abstract interpretation, a technique originally devised for compiler verification but most recently applied to labeled transition systems as a solution to the state-explosion problem in model checking of reactive systems [SS99]. The idea is to minimize the number of possible states of a labeled transition system by introducing predicates describing the possible configurations of a control state. The abstract representation then contains just boolean variables representing the original predicates of the concrete system. A so-called boolean abstraction of the predicate can then be calculated algorithmically. As the system is viewed as a predicate transformer, rather than a transition system, the abstraction of the concrete predicates together with a suitable abstraction of the conditions at the labels, may represent a faithful abstraction, *i.e.* an abstraction that preserves the properties of the original system. Although in principle this abstraction process is sound it also depends on the quality of the predicates chosen for the abstraction, whether it is property preserving or not. The technique does not provide a systematic way of finding the right predicates.

This technique of boolean abstraction is well-established and algorithms for the efficient computation are known and implemented, *e.g.* in PVS [SS99]. To apply the algorithms to our representation of statecharts by HA we have to flatten the structure and resolve the parallelism, in order to get a classical labeled transition system needed as input. This transformation can simply be performed by first flattening the hierarchy using name-spacing and then producing a product automaton.

However, the problem for our application is that — as in general for this abstraction technique — we need to find the right level of abstraction. In particular, for the class of systems we are considering, the racing conditions need to be preserved in order to have an adequate abstraction. Considering an arbitrary boolean abstraction it is obvious that racing conditions may just vanish in the abstract representation: consider that by the abstraction the assignments `y := 1` and `y := 4` are mapped to the same abstract state, because the predicate used for the abstraction are, say, $\lambda(x, y).\ y \leq 0$ and $\lambda(x, y).\ y > 0$

The hand-made abstraction we use as an input to the SMV system is only partly adequate as long as the data stay inside the assumed boundaries. It does not apply the above described property based abstraction technique. However, inside the bounds the racing conditions are preserved.

As a systematic way of describing how boolean abstractions of a system may be found in general and in particular with respect to the racing problems at hand consider the following criteria.

- The granularity of the predicates used as input to the abstraction has to respect the ranges of racing assignments. With respect to the previous example this would imply that the predicates have to put `1` and `4` in different classes.
- The predicates may well be inferred from preconditions of control states. More precisely, the abstraction predicates may be derived as the disjunctive normal form of the postconditions of incoming and preconditions of outgoing transitions of a control state.

The first criteria is a necessary condition to preserve racing condition. The second criteria is a basis for a systematic derivation of suitable predicates for an abstraction, a question so far not part of the known abstraction techniques, and generally applicable as a preparatory step to abstraction. Another necessary condition for the automatic calculation of abstraction predicates according to criteria two is to preserve the preconditions of racing transitions. This can be achieved by a further separation of the derived predicates according to the preconditions of the outgoing transitions that entail the racing assignments.

The application of existing abstraction techniques necessitates the dissolution of hierarchy and parallelism of HA. As a future extension for the abstraction methods we consider lifting the known techniques to HA.

8 Conclusions

The combination of model checking and theorem proving techniques is a widely examined field. To focus this combination on particular formalisms has more recently found greater interest because the integration of the two analysis techniques benefits from restriction to a specific application domain, like labeled transition systems or statecharts. The paper [SS99] is related to our work, but it is about labeled transition system rather than statecharts. Although statecharts may be flattened to labeled transition systems — an aspect that comes in handy when thinking about applying the described abstraction techniques — we believe statecharts to be a worthwhile object of study as their additional structure and parallelism is a powerful mechanism for system development. Concerning verification of statecharts using higher order logic theorem proving combined with model checking the work by Day [Day93] is probably most relevant to our work, but it only deals with simple unstructured data spaces and does not address racing conflicts. Another relevant formalization is the work by Nipkow and Slind [NS95] on I/O-Automata, a formalism differing from statecharts. For example, I/O-automata have no hierarchy. The formalization does not consider model checking aspects but gives a thorough treatment of the meta-theory of I/O-automata. Other combination of higher order logic theorem proving and model checking for reactive system is the work by Nipkow and Müller [MN95] again on I/O-automata. In principle it addresses the question of data spaces in a reactive system but assumes data independence which assumes that the data spaces contained in a reactive system can be ignored. This is an aspect that

is the focus of the current work as we consider racing conditions, a phenomen showing that data independence is a fairly unrealistic assumption.

This work has contributed to the mechanical analysis of statecharts in two ways. First, we show how data spaces may be easily integrated into the representation of hierarchical automata by extending the states and labels. Furthermore, this extension by data spaces is constructed in such a way that the resulting problem of racing conditions may be resolved in a simple way using features of HOL. Secondly, we show how the HA with data can be effectively transformed into a representation for the model checker SMV. Here, the racing resolution can also be achieved by introducing disjunctions of next states. The necessary reduction of the state space introduced by infinite data structures has only been achieved by imposing finite domains. This simplistic view has been taken to preserve properties, in particular the racing conditions. However, we also presented a description of how to adjust well-known abstraction techniques such that they can be applied to statecharts with data spaces in a property preserving way. We also sketched how to derive the necessary predicates needed as input to the abstraction process.

References

[BG97] R. Büssow, W. Grieskamp. Combining Z and temporal interval logics for the formalization of properties and behaviors of embedded systems. In *Asian Computing Science Conference (ASIAN'97), Springer LNCS*, **1345**:46–56, 1997.

[BW98] U. Brockmeyer, G. Wittich. Tamagotchis need not die – verification of statemate design. In B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'98, Springer LNCS*, **1384**, 1998.

[BG98] R. Büssow, W. Grieskamp. The ZETA System Developer's Guide Technische Universität Berlin, December 1998. www.uebb.cs.tu-berlin.de/zeta/.

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

[Day93] Nancy Day. A model checker for statecharts. Technical Report TR-93-35, Department of Computer Science, University of British Columbia, October 1993.

[Hie98] J.-J. Hiemer. Statecharts in CSP – Ein Prozessmodell in CSP zur Analyse von STATEMATE Statecharts. *PhD thesis, Technische Universität Berlin*, 1998.

[HN96] David Harel and D. Naamad. A STATEMATE semantics for statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, Oct 1996.

[KH00] F. Kammüller and S. Helke. Mechanical Analysis of UML State Machines and Class Diagrams. In *Defining Precise Semantics for UML*, Sophia Antipolis, France, June 2000. Satellite Workshop, ECOOP 2000.

[HK01] S Helke and F. Kammüller. Representing Hierarchical Automata in Interactive Theorem Provers. In R. J. Boulton, P. B. Jackson, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2001, Springer LNCS*, **2152**, 2001.

[McM93] K. McMillan. Symbolic Model Checking: An Approach to the State Explosion Problem. Kluwer Academic Publishers, 1993.

[Mikk00] E. Mikk. Semantics and Verification of Statecharts. *PhD thesis, Christian Albrechts Universität Kiel*, 2000.

[MLS97] E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical automata as model for statecharts. In *Asian Computing Science Conference (ASIAN'97)*, *Springer LNCS*, **1345**, 1997.

[MN95] Olaf Müller and Tobias Nipkow. Combining Model Checking and Deduction for I/O Automata. In *TACAS'95, 1st International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, *Springer LNCS*, **1019**:1–16, 1995.

[MSS86] A. Melton, D. A. Schmidt and G. E. Strecker. Galois connections and Computer Science applications, *Springer LNCS*, **240**:299–312, 1986.

[NS95] T. Nipkow and K. Slind. I/O Automata in Isabelle/HOL. In Types for Proofs and Programs, *Springer LNCS*, **996**:101–119, 1995.

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, *Springer LNCS*, **828**, 1994.

[SS99] Hassen Saïdi and Natarajan Shankar. Abstract and Model Check While You Prove. In N. Halbwachs and D. Peled, editors, *11th Interantional Conference on Computer Aided Verification, CAV'99*, *Springer LNCS*, **1633**, 1999.

Formalization and Execution of STE in HOL

Ashish Darbari

Computing Laboratory
University of Oxford
Oxford, England, OX1 3QD

Abstract. We present an early implementation of STE model checking in the higher-order logic theorem prover `HOL`. Our results are based on an earlier work done by [1, 2] in combining STE with theorem proving. By way of formalizing the results presented in [1], we have an initial platform for executing STE semantics directly in `HOL`. We can relate correctness results of the STE logic to the Boolean logic of `HOL`. We show how any trajectory assertion that is validated to true in STE, can be translated to an equivalent theorem in `HOL`. To this end we have extended the work presented in [1, 2] by implementing not only the theoretical results of [1, 2] but also incorporating the core STE implementation presented in [3]. As a useful benefit of proving the lemmas and theorems on machine, we discovered a flaw in the proof of one of the lemmas presented in [2].

1 Introduction

One of the main challenges posed to the verification engineers today is to manage the size of the verification problem. Classic verification techniques like symbolic model checking typically suffer from state explosion problem. However the degree to which they allow automation, and the expressivity of the language of the model checker, makes them very useful for verifying complex temporal properties.

Deductive verification techniques like theorem proving can handle a verification task of any size, but at the cost of manual intervention. Even the very best state-of-the art theorem provers require substantial manual guidance throughout the proof. To overcome the limitations of each of the above verification approaches, the idea is to blend them together to exploit the strength of each one of them, and alleviate the weaknesses. The work presented in this paper falls in the general area of combining model checking with theorem proving. Specifically we are investigating the combination of symbolic trajectory evaluation based model checking with higher order logic theorem proving.

Symbolic trajectory evaluation [3] or STE in short, is a highly effective model checking technique for datapath verification [4]. It has been combined with theorem proving to verify complex industrial designs [5, 6].

Aagaard et.al. in [1] outlined the theoretical foundation for linking the general logic of STE with higher order logic. They outlined the issues involved in making such a combination, and then presented a cohesive theory of integration, proving lemmas and theorems which justify a sound semantic link between STE and theorem proving. However, in that paper they did not provide formal proofs of the lemmas and the theorems. In the extended technical report [2] they provided proofs of some of the lemmas and theorems on paper. They claimed in their paper and the report that using the lemmas and theorems we can in principle verify properties using STE model checking and deduce the equivalent theorems in higher order logic. They however did not give any implementation details to show how this can be achieved in practice.

In this paper we provide the implementation machinery for integrating STE model checking with higher order theorem proving based on the results presented in [1, 2]. We show in this paper the implementation details of embedding the STE logic in a higher order theorem prover HOL.[1] By having an implementation of the theory of STE, and the semantic link to higher order logic, we are able to execute STE directly in HOL. Our implementation allows us to check properties using STE, and if the property is valid, we get an equivalent theorem in HOL, thereby achieving exactly what was envisioned by Aagaard et.al. in [1, 2]. We have tested a few examples using our implementation and the initial results are encouraging.

As a side benefit of mechanizing the theory presented in [1, 2], we discovered that there is a discrepancy in the proof of one of the lemmas in [2]. We shall talk more about it when we show the mechanized version of that lemma.

We are not able to present in this paper, the machine proofs and the example because of lack of space. The script file with all the details is available online.[2]

1.1 Related Work

Many researchers in the past have considered this problem of integrating the model checking with theorem proving [5, 7–9].

Rajan et.al. [7] have presented an integration of a BDD based model checker for propositional μ-calculus with the PVS theorem prover. They argued that μ-calculus serves as a good basis for combining model checking with theorem proving.

The long term goal of our research is to combine STE with theorem proving for verifying large circuit designs. We have been greatly inspired by the work done by Aagaard et.al. [5]. They claim that their verification effort resulted in the discovery of eight previously unknown bugs, four of which were high quality bugs – meaning they would not have been diagnosed with traditional validation techniques.

The work we are presenting in this paper however comes closest to [1, 2, 8, 10].

[1] HOL here stands for the theorem prover HOL 4, Kananaskis 1

[2] http://users.comlab.ox.ac.uk/ashish.darbari/Research/TPHOLS03/

Joyce and Seger in the past have worked on combining the Voss system with the HOL theorem prover [8]. They focused on Voss specific implementation of STE. We took the approach advocated by Aagaard et.al. [1,2] in actually integrating the general logic of STE with the HOL theorem prover.

Aagaard et.al. in [10] proposed a solution of combining STE with theorem proving using a strongly typed functional language in the ML family, called fl. They lifted the language to make it reflective similar to Lisp. This gives them a possibility of executing fl functions and also to reason about the behavior of fl functions. The link from theorem proving to model checking is established in their approach by evaluating the lifted fl expressions.

The theorem proving support they offer in the lifted-fl language is an LCF-style implementation. However the core of the theorem prover is a set of trusted tactics and is not not fully expansive. Tactics work backwards and do not allow forward proofs.

1.2 Organization of the Paper

In this section we outline the road map of our paper. We start in the next section by presenting an outline of the basic STE theory. We show fragments of our implementation of the definitions of the basic concepts of the STE theory in HOL. In Section 3 we present the formalization of the link between STE and the two valued Boolean logic, providing relevant details together with the lemmas and theorems which justify the link.

In Section 4 we show how to use the combined formalization of STE theory and the linkage with HOL, to execute STE on an example design. In the last section we present conclusions and point to future directions.

2 Symbolic Trajectory Evaluation

Symbolic trajectory evaluation [3], combines the ideas of *ternary modelling* with *symbolic simulation*. In ternary modelling the binary set of values {0, 1} is extended with a third value X which indicates an unknown logic value. By assuming a monotonicity property of the simulation algorithm one can ensure that any binary value resulting when simulating patterns containing X's would also result when X's are replaceed by 0's and 1's. Thus the number of patterns that must be simulated to verify a circuit are reduced dramatically by representing many different operating conditions by patterns containing X's. With ternary simulation, a state with some nodes set to X covers those circuit states obtained by replacing the X values with either a 0 or a 1. The state with all nodes set to X thus covers all possible actual circuit states.

Although ternary modelling allows us to cover many conditions with a single simulation run, it lacks the power required for complete verification, except for a small class of circuits such as memories [11].

Symbolic trajectory evaluation extends the idea of ternary modelling by including the notion of time and the usage of symbolic Boolean variables. Using

STE one can specify and verify system behavior over time. By using symbolic Boolean variables and propositional logic expressions over these, we can represent whole classes of data values on circuit nodes. The Boolean expressions or BDDs respresenting values at different circuit nodes can have variables in common. These variables can record complex interdependencies among node values.

In the subsequent sections, we shall present an implementation of STE in HOL. Readers unfamiliar with the detailed theory of STE and the syntax of HOL are referred to [1, 3].

2.1 The four valued lattice

Symbolic trajectory evaluation employs a ternary circuit state model, in which the usual binary values 0 and 1 are augmented with a third value X that stands for an unknown. To represent this mathematically, we introduce a partial order relation $\sqsubseteq$, with $X \sqsubseteq 0$ and $X \sqsubseteq 1$. The relation orders values by information content: X stands for a value about which we know nothing, and so is ordered below the specific values 0 and 1.

To develop a smooth mathematical theory for STE, we add a further value $\top$ (called 'top') to get the set of circuit node values $\mathcal{D} = \{0, 1, X\} \cup \top$. We extend the ordering relation to make $(\mathcal{D}, \sqsubseteq)$ a complete lattice.
We use the idea of dual-rail encoding [12], to define the four lattice values in the STE logic.

```
(* Four lattice values *)
 |- Top = (F,F)
 |- One = (T,F)
 |- Zero = (F,T)
 |- X = (T,T)
```

Please note that the choice of Top, being defined as (F, F) and other values like X being (T, T) is in principle arbitrary, any possible permutation on the two Boolean values T, and F can be chosen to denote the four values. However efficient definition of least upper bound, and the ordering $\sqsubseteq$, does depend crucially on a particular permutation of T and F that we choose for representing the lattice values.

```
(* Least upper bound (lub) *)
 |- ∀a b c d. lub (a,b) (c,d) = (a ∧ c,b ∧ d)

(* Information ordering *)
 |- ∀a b. leq a b = (b = lub a b)
```

Observable points in circuits are nodes. Nodes can be defined by the HOL type string. A lattice state is then defined as an instantaneous snapshot of circuit behavior given by an assignment of lattice values to nodes. If we denote the lattice state by s then

```
s: string->(bool # bool)
```

A lattice sequence assigns a lattice value to each node at each point in time. Time is just the set of natural numbers (num in HOL). If we denote the lattice sequence by sigma then

```
sigma: num->string->(bool # bool)
```

Because of lack of space here, we do not show all the functions that we have implemented in HOL. However, we do think its important to mention them because we use them in other definitions. Some of these functions we wrote in HOL are the Suffix and the extension of the information ordering on lattice states (leq_state) and sequences (leq_seq).

2.2 Circuit Models in STE

In STE, the formal model of a circuit is given by a next-state function Y_ckt that maps lattice states to lattice states:

```
Y_ckt: (string->(bool # bool))->(string->(bool#bool))
```

Intuitively, the next-state function expresses a constraint on the set of possible states into which the circuit may go for any given state. In implementations of STE, the circuit model Y_ckt is constructed incrementally and piecemeal by ternary symbolic simulation of an HDL or a netlist source for the circuit. In our presentation here the circuit is uninterpreted. To run a typical example using our STE formalization requires one to define the model Y_ckt completely.[3]

One crucial property the next-state function needs to preserve is the property of monotonicity. Any next-state funcntion is monotonic if for all lattice states s and s', if s $\sqsubseteq$ s' then state obtained by applying the next-state function (Y_ckt) on s is also less than or equal to ($\sqsubseteq$) the state reached by applying Y_ckt to s'.

Since sequences return lattice values for each node at a given time point, a sequence encodes a set of behaviors that a circuit can exhibit. The next-state function or the lattice model of the circuit provides the meaning of circuit behavior. Now we shall define what it means for a sequence to be in the lattice model of a circuit. A sequence is in the language of a lattice model of a circuit if the set of behaviors that the sequence encodes is a subset of the beahviors that the circuit can actually exhibit. Below we show the formalization in HOL.

```
(* Lattice sequence in the language of a circuit *)
 |- ∀sigma Y_ckt.
       in_STE_lang sigma Y_ckt =
       ∀t. leq_state (Y_ckt (sigma t))(sigma (t + 1))
```

[3] We will show in a later section, how we do this for a specific circuit.

2.3 Syntax of STE

In STE, the basic syntactic entity used in specifcation is a symbolic trajectory formula. In formalizing the definition of STE syntax, we define a new type `TF`, of trajectory formulas in `HOL`.

```
(* Syntax of trajectory formula *)
 val _ = Hol_datatype
   `TF =
   Is_0 of string
 | Is_1 of string
 | AND of TF => TF
 | WHEN of TF => bool
 | NEXT of TF`;
```

We have a deep embedding [13–15] of all the operators `Is_0`, `Is_1`, `AND`, `WHEN` and `NEXT`. However we have chosen to represent the guard [1, 2] shallowly by actually using the `HOL` type `bool` to represent the guard.[4] This is to allow us to inherit the theory of booleans (`bool`) in `HOL`. The HolBdd [16] package is also interfaced to the type `bool` in `HOL`, possibly later we can use the HolBdd package, without having to invest too much effort (we won't have to reinvent the theory of Booleans and link them with BDDs).

2.4 Semantics of STE

We now define the semantics of the trajectory formula in `HOL`. We formalize in `HOL`, the function `SAT_STE`, that defines when a trajectory formula is satisfied by the lattice sequence.

```
(* Sequence satisfying a formula *)
 |- (∀n. SAT_STE (Is_0 n) = (λsigma. leq Zero (sigma 0 n)))
 ∧ (∀n. SAT_STE (Is_1 n) = (λsigma. leq One (sigma 0 n)))
 ∧  (∀tf1 tf2. SAT_STE (tf1 AND tf2) = (λsigma. SAT_STE tf1 sigma
                                        ∧ SAT_STE tf2 sigma))
 ∧ (∀tf P.SAT_STE (tf WHEN P) = (λsigma. P ==> SAT_STE tf sigma))
 ∧ (∀tf. SAT_STE (NEXT tf) = (λsigma. SAT_STE tf (Suffix 1 sigma)))
```

[4] Angelo et.al. in [15] and Boulton et.al. in [14] present interesting case studies of different kinds of embeddings of hardware description languages.

2.5 STE Verification Engine

Verification in STE takes place by testing the validity of an assertion of the form

```
Ant ==>> Cons
```

where `Ant` and `Cons` are trajectory formulas having the abstract type `TF` in `HOL` formalization, and `==>>` is a constructor that takes two elements of type `TF` and returns an element of an abstract type `Assertion` in `HOL`.

The function that checks the validity of such an assertion is formalized in `HOL` as `SAT_CKT`

```
(* Validity of a trajectory assertion *)
 |- ∀Ant Cons Y_ckt.
       SAT_CKT (Ant ==>> Cons) Y_ckt =
         ∀sigma. in_STE_lang sigma Y_ckt ==>
                  ∀t. SAT_STE Ant  (Suffix t sigma)  ==>
                                SAT_STE Cons (Suffix t sigma)
```

Seger and Bryant in [3] proposed an implementation algorithm of STE. They introduced the idea of a defining sequence and a defining trajectory. They then argued that any trajectory assertion of the form `Ant ==>> Cons` can be verified by the STE implementation if and only if the defining sequence of `Cons` is less than or equal to ($\sqsubseteq$) the defining trajectory of `Ant`, for all nodes mentioned in the assertion and for all time points upto the depth of `Cons`.

We have defined the functions to calculate the defining sequence (`DefSeq`) and the defining trajectory (`DefTraj`) in `HOL`, unfortunately we cannot show their formalized definition here due to lack of space.

We now state the STE implementation (`STE_Impl`) algorithm [3], that takes an assertion and a circuit model `Y_ckt` and computes a symbolic constraint (over the free variables appearing in the guard of the trajectory formulas in the assertion) under which the assertion will be valid. The strength of this implementation algorithm lies in the fact that it is sufficient to compute finite segments of the defining sequence and the defining trajectory, to completely verify the assertion even though in theory both the defining sequence and the defining trajectory is infinite. The depth of the segment is computed from the depth of the consequent in the assertion.

```
(* STE implementation *)
 |- ∀Ant Cons Y_ckt.
       STE_Impl (Ant ==>> Cons) Y_ckt = ∀t. t <= Depth Cons ==>
             ∀n. MEMBER n (Nodes Ant Append Nodes Cons) ==>
                        leq (DefSeq Cons t n) (DefTraj Ant Y_ckt t n)
```

The function `Depth` calculates the number of `NEXT` operators in a trajectory formula. The function `Nodes`, calculates the list of nodes in a given formula, and

the function MEMEBR checks for the occurence of a node in a given list of nodes. Append is the usual append on lists. We have defined all these functions in HOL.

We now present the theorem[5] that makes an assertion about the correctness of the STE algorithm.

The theorem states that the trajectory assertion is valid for a circuit with model Y_ckt if and only if the STE implementation guarantees that the trajectory assertion is valid for the model Y_ckt.

```
(* Theorem 1: Correctness of STE algorithm *)
|- ∀Ant Cons Y_ckt.
       SAT_CKT (Ant ==>> Cons) Y_ckt
                     = STE_Impl (Ant ==>> Cons) Y_ckt
```

3 From lattice world to the relational world

The language of the theorem prover HOL is based on a Boolean logic. Hence in order to make a connection between STE and HOL, we have to address the key problem of connecting the four valued STE logic to a two-valued Boolean logic. This entails addressing the following issues

- defining when the embedded STE trajectory formulas are satisfied by a Boolean valued sequence
- defining a connection between the lattice values and the Boolean values
- identifying a connection between the circuit model in STE world and the circuit model in the Boolean world
- relating correctness results in the STE world to correctness results in the Boolean world

We shall discuss these issues in subsequent sections.

3.1 Semantics of Trajectory Formulas in Boolean Logic

States in the Boolean world are functions from the set of nodes $\mathcal{N}$ to the Boolean set $\mathcal{B}$, where $\mathcal{B} = \{\mathtt{T}, \mathtt{F}\}$. We refer to the states in the Boolean world as *Boolean states*. The set $\mathcal{B}$ is the set bool in HOL. Using the type string to denote the set of nodes $\mathcal{N}$ we shall represent a Boolean state by subscripting the letter s with b.

```
s_b: string->bool
```

A *Boolean sequence* is a function which returns a Boolean state, at given point of time. We denote the Boolean sequence by sigma_b in HOL and time is denoted by the type num.

```
sigma_b: num->string->bool
```

[5] At present we have used this theorem as an axiom in HOL, since we are not finished with the proof yet.

We shall now define the function SAT_BOOL that defines when a trajectory formula is satisfied by a Boolean sequence sigma_b.

```
(* Boolean sequence satisfies a trajectory formula *)
 |- (∀n. SAT_BOOL (Is_0 n) = (λsigma_b. sigma_b 0 n = F))
 ∧ (∀n. SAT_BOOL (Is_1 n) = (λsigma_b. sigma_b 0 n = T))
 ∧ (∀tf1 tf2.
       SAT_BOOL (tf1 AND tf2) =
          (λsigma_b. SAT_BOOL tf1 sigma_b ∧ SAT_BOOL tf2 sigma_b))
 ∧ (∀tf P.
       SAT_BOOL (tf WHEN P) = (λsigma_b. P ==> SAT_BOOL tf sigma_b))
 ∧ (∀tf.
       SAT_BOOL (NEXT tf) =
         (λsigma_b. SAT_BOOL tf (Suffix_b 1 sigma_b)))
```

The function Suffix_b is defined in a way similar to the function Suffix. Suffix_b returns the i^{th} suffix of a Boolean sequence.

3.2 Relating Lattice values to Boolean values

We define an operation called drop which drops the values from the Boolean world to the values in the STE world.

```
(* Dropping from Boolean to lattice Values *)
 |- (drop T = One) ∧ (drop F = Zero)
```

We shall need the point wise extension of the drop operation on states and sequences, in order to define some useful lemmas later. Lifting the drop operation pointwise, we can relate the lattice valued states and sequences to the Boolean valued states and sequences as

```
(* Drop operation lifted over states *)
 |- ∀s_b. extended_drop_state s_b =  (λnode. drop (s_b node))
(* Definition : Drop operation lifted over sequences *)
 |- ∀sigma_b.
       extended_drop_seq sigma_b =
                     (λt. extended_drop_state (sigma_b t))
```

3.3 Relational Circuit Model

A circuit in the Boolean world, is modelled by a next-state relation, which for a given circuit gives a relation between present and next Boolean state. The circuit is uninterpreted here similar to the way it was in the definition of the lattice model. While running example circuits, we define the relational model of a circuit in HOL. We will show in a later section how we accomplish this for a concrete example.

```
Yb_ckt: (string->bool)->(string->bool)->bool
```

3.4 Boolean Sequence in the language of the circuit

A Boolean valued sequence is in the language of the circuit, with the relational model `Yb_ckt`, iff the consecutive Boolean valued states are included in the next-state relation `Yb_ckt`.

```
(* Boolean sequence is in the language of a circuit *)
 |- ∀sigma_b Yb_ckt.
        in_BOOL_lang sigma_b Yb_ckt =
        ∀t. Yb_ckt (sigma_b t) (sigma_b (t + 1))
```

3.5 Relating circuit models in STE and Boolean World

We have made connections between Boolean and lattice valued states and sequences. In order to make a sound connection between the functional circuit model in the STE world with the relational circuit model of the Boolean world, we need to make sure that the two models of the circuit (`Y_ckt` and `Yb_ckt`) describe the same behavior.

Intuitively, for a given circuit, with a relational model `Yb_ckt` and the lattice model `Y_ckt`, the two circuit models describe the same circuit, if and only if for any two Boolean states `s_b` and `s_b'` (where `s_b` is the present state and `s_b'` is the state at next point of time) if `s_b` and s_b' are related by the relational model `Yb_ckt`, then the lattice model `Y_ckt` when applied to the drop of the present Boolean state `s_b` should return a lattice value, that conveys information less than or equal ($\sqsubseteq$) to, the information conveyed by the lattice value returned by the drop of the next Boolean state `s_b'`.

We define the predicate `Okay` in `HOL`, that asserts when the two circuit models describe the same circuit.

```
(* Linking Boolean and lattice models *)
 |- ∀Y_ckt Yb_ckt.
        Okay (Y_ckt,Yb_ckt) =
        ∀s_b s_b'.
          Yb_ckt s_b s_b' ==>
          leq_state (Y_ckt (extended_drop_state s_b))
                    (extended_drop_state s_b')
```

3.6 Relating Correctness Results

In this section we shall relate the correctness results from the STE world to the Boolean world. The intuition is that any trajectory assertion that is satisfied by lattice valued sequence should be satisfied by the Boolean valued sequence.

Before we state the theorem that relates the correctness results between the two worlds, we shall state two lemmas which we have used in the proof of the theorem.

```
(* Lemma 1: Relating Boolean and lattice valued sequences *)
 ∀Y_ckt Yb_ckt.
          Okay (Y_ckt, Yb_ckt) ==>
            ∀sigma_b. in_BOOL_lang sigma_b Yb_ckt ==>
                        in_STE_lang (extended_drop_seq sigma_b) Y_ckt
```

The lemma states a fact that whenever the two circuit models `Y_ckt` and `Yb_ckt` talk about the same circuit (i.e. satisfy the property `Okay`) then for every Boolean sequence which is in the relational model of the circuit, the drop of the Boolean sequence is in the lattice model of the circuit.

As mentioned earlier in the introduction, the proof of Lemma 1 presented in [2] is incorrect. The lemma as stated in [2], says that whenever two circuits satisfy the property `Okay` (Axiom 2 in [2]), then every Boolean sequence is in the language of the relational model of the circuit if and only if the drop of the Boolean sequence is in the lattice model of the circuit. The proof of the lemma relies on Axiom 2 (in [2]), which is stated as an implication. Just by using the implication in Axiom 2, we cannot prove the equivalence property of the lemma [2].

Our claim here is that either we state both Axiom 2 and Lemma 1 (in [2]) as an implication or state them both as an equivalence. We chose to keep an implication in the definition of `Okay` (the counterpart of Axiom 2), since it gives us enough power to say what we wanted to say, and we also state `Lemma 1` (the counterpart of Lemma 1 in [2]) as an implication. Interestingly, in the paper [1] the authors have stated the Axiom and the Lemma both as an implication.

We now state a lemma below which captures the fact that a trajectory formula is satisfiable by a Boolean sequence if and only if it is satisfiable by the drop of the Boolean sequence.

```
(* Lemma 2: Relating satisfaction over Boolean and lattice valued
            sequences *)
 ∀tf seq_b. SAT_BOOL tf seq_b = SAT_STE tf (extended_drop_seq seq_b)
```

Now we are in a position to state `Theorem 2`. `Theorem 2` states that for a given circuit with lattice model `Y_ckt` and the Boolean model `Yb_ckt`, if `Y_ckt` and `Yb_ckt` satisfy the property `Okay`, then if a given trajectory assertion is satisfied by the lattice model, then for all Boolean valued sequences which are in the language of the Boolean model `Yb_ckt`, for all time points `t` greater than zero, if the antecedent of the trajectory assertion (`Ant`) is satisfied by the $\mathtt{t}^{th}$ suffix of the Boolean valued sequence, then the consequent of the trajectory assertion `Cons` is also satisfied by the $\mathtt{t}^{th}$ suffix of the Boolean valued sequence.

```
(* Theorem 2: Correctness in STE world implies correctness
              in the Boolean world *)

|- ∀Ant Cons Y_ckt Yb_ckt.
       Okay (Y_ckt,Yb_ckt) ==>
       SAT_CKT (Ant ==>> Cons) Y_ckt ==>
       ∀sigma_b.
         in_BOOL_lang sigma_b Yb_ckt ==>
         ∀t.
           SAT_BOOL Ant (Suffix_b t sigma_b) ==>
           SAT_BOOL Cons (Suffix_b t sigma_b)
```

`Theorem 2` forms the crux of the connection between the lattice world and the relational world. It gives us the power to link the correctness statements in STE world to the notion of correctness in the relational world. This means we can use the STE verification engine to compute a symbolic constraint under which a trajectory assertion would be valid, and then infer a corresponding Theorem in the relational world. These theorems in the Boolean world are the theorems we intuitively expect to hold when the property stated in the trajectory assertion is verified independently in the theorem prover.

Of course, the validity of `Theorem 2` and `Lemma 1` relies on the fact that it is possible to translate the correctness statements from the STE world to the Boolean world only if the circuit models in the two worlds satisfy the property `Okay`.

In the next section we show how we combine `Theorem 1` and `Theorem 2` to prove for the unit-delay Nand gate[6] that if an implementation of STE algorithm (`STE_Impl`) returns the value `T`, then we can get an equivalent theorem in `HOL`.

4 Executing STE in HOL

In this section we illustrate the example of a two input unit-delay Nand gate, whose output is tied to one of its inputs (see [1, 2]). We define the lattice and the Boolean models for such a circuit. Below is an example we wrote in `HOL`.

[6] We have taken the example from [1, 2]

```
(* Definition of Not, And and Nand using dual-rail encoding *)
|- ∀a b. Not (a, b) = (b, a)
|- ∀a b c d. And (a, b) (c, d) = (a ∧ c, b ∨ d)
|- ∀a b. Nand a b = Not (And a b)

(* Definition of lattice model for the unit delay Nand gate *)
|- ∀s node.
       Nand_lattice s node =
       (if node = "in" then
          X
        else
          (if node = "out" then Nand (s "in") (s node) else X))

(* Definition of the relational model for the unit delay Nand gate *)
|- ∀s_b s_b'.
       Nand_bool s_b s_b' =
       ∀node.
         ((node = "out") ==> (s_b' node = ~(s_b "in" ∧ s_b node))) ∧
         ((node = "in") ==> s_b' node ∨ ~s_b' node)
```

We then write the STE assertions that we need to verify, using the STE implementation `STE_Impl`. Intuitively, if we assert the Boolean variables `v1` and `v2` on one of the input nodes and the output node respectively, then after one unit of time we can expect to observe the value $\neg$(`v1` $\wedge$ `v2`) at the output. Infact this is exactly what we assert in the STE assertion as shown below.

```
(* input node has a value v1 *)
val ant1 = ``(Is_1 "in" WHEN v1) AND (Is_0 "in" WHEN ~v1)``

(* output node has a value v2 *)
val ant2 = ``((Is_1 "out") WHEN v2) AND ((Is_0 "out") WHEN ~v2)``

(* Antecedent: "in" is v1 and "out" is v2 *)
val Ant = Term `^ant1 AND ^ant2`;;

(* Consequent: N("out" is ~(v1∧v2)) *)
val Cons = ``NEXT
      ((Is_1 "out" WHEN ~(v1 ∧ v2)) AND (Is_0 "out" WHEN (v1 ∧ v2)))``
```

We have written ML functions and (conversions[7] in `HOL`) to develop automated proof strategies which perform computation. Here we will present informally an outline.

[7] conversions have the type `term -> thm`

Theorem 1 states the equivalence of SAT_CKT and the STE_Impl. We substitute STE_Impl for SAT_CKT in Theorem 2 and we get an auxiliary theorem, that relates the STE_Impl and the satisfaction of trajectory assertion over Boolean sequence. We then take this auxiliary theorem and apply some of our hard-wired conversions on it, to eventually get the desired theorem in HOL.

We wrote the top-level function STE_TO_BOOL that takes an antecedent, a consequent, the lattice model of the circuit (‘‘Nand_lattice‘‘), the relational model (‘‘Nand_bool‘‘) and the string "Nand" and it computes a theorem in HOL. The string "Nand" actually tells the function STE_TO_BOOL to use the conversion written specifically for the Nand gate circuit.

```
- STE_TO_BOOL Ant Cons ‘‘Nand_lattice‘‘ ‘‘Nand_bool‘‘ "Nand";
  runtime: 14.100s,    gctime: 2.070s,     systime: 0.050s.
  Meson search level: ................................
> val it =
|- ∀v2 v1 sigma_b.
    in_BOOL_lang sigma_b Nand_bool ==>
    ∀t.
     (sigma_b t "in" = v1) ∧ (sigma_b t "out" = v2) ==>
     (sigma_b (t + 1) "out" = ~(sigma_b t "in" ∧ sigma_b t "out"))
```

In the above example the trajectroy assertion is true, for any assignment of Boolean values to the variables v1 and v2. So the STE implementation in this case returns the value T.

If the STE implementation doesn't return the value T but instead returns a symbolic Boolean expression (residual), even then we can get an equivalent theorem in HOL, however that theorem will have the residual as an assumption. We are at present working on developing functions that will take the residual and come up with a counter examples or sets of satisfying valuations that will make the residual T. One possibility we are considering is to use the HolSatLib [17] package. At the moment we have not completely investigated this, but this definitely something for future work.

The functions and conversions that we have written have two components, one is a fairly general component that can take any circuit model and do some pre-processing; the other specific component is tailored to handle the proof of the Okay property for each specific circuit in question. Since the proof for each circuit in question depends on the model definitions, it seems impractical to have one general purpose proof routine for every circuit.

5 Conclusion and Future Work

In this paper we presented the formalization of the STE logic in HOL. We formalized the results presented in Aagaard et.al's work [1] on linking trajectory evaluation to higher-order logic. We also extended their idea by writing functions that implement the core STE algorithm known from [3] and show that one

can execute the semantics of STE directly in a theorem prover like HOL. In this process, we wrote special purpose proof strategies (conversions) that we used to advance the computation of the STE Implementation and reach to a point where we get theorem in HOL.

Since our work is in a preliminary stage, we cannot yet compare our implementation with Aagaard et.al. [10]. It seems it will be very useful to compare and also draw on their experience of doing a similar task, specially because they use a language (lifted-fl) specially tailored for this kind of task. The language allows representation of Boolean expressions as BDDs. This gives them a seamless integration of model checking and theorem proving. In our case, we don't model guards in STE by BDDS.

In HOL the Booleans and the BDDS are two different types. We will need to stitch them together possibly using the HolBdd package [16]. At the moment we have not completely investigated the usage of HolBdd and the ramifications it will have on our work. This is one of the goals we have set for immediate future work in this area. Together with BDDs we intend to experiment with other abstraction ideas which can help us reduce the verification effort of large circuit designs.

We are also working on making the function STE_Impl more efficient, and optimizing other functions and conversions that we have.

In the process of formalizing the theory of STE we have uncovered a bug in the proof of one of the lemmas stated in the technical report [2]. Although the discrepancy isn't a major bug in the report, we believe our effort in uncovering it is well worth it.

6 Acknowledgment

Thanks are due to Mike Gordon for arranging my visit to the Computing Laboratory at Cambridge and motivating me to work on this. Michael Norrish at Cambridge provided me some useful hints on how to get started with HOL. John O' Leary, Jim Grundy and Robert Jones at Intel, gave me useful feedback in early stages of my work. Myra VanInwegen gave useful feedback on an early draft of the paper.

Special thanks to Tom Melham who really made a difference with his expertise in the subject and has provided continuous help and inspiration.

This work has been supported in part by a research grant from Intel Corp. USA.

References

1. M. D. Aagaard, T. F. Melham, and J. W. O'Leary, "Xs are for trajectory evaluation, Booleans are for theorem proving," in *Correct Hardware Design and Verification Methods: 10th IFIP WG10.5 Advanced Research Working Conference: Bad Herrenalb, September 1999: Proceedings*, L. Pierre and T. Kropf, Eds. 1999, vol. 1703 of *Lecture Notes in Computer Science*, pp. 202–218, Springer-Verlag.

2. M. D. Aagaard, T. F. Melham, and J. W. O'Leary, "Xs are for trajectory evaluation, Booleans are for theorem proving (extended version)," Tech. Rep. TR-2000-52, Department of Computing Science, University of Glasgow, January 2000.
3. C.-J. H. Seger and R. E. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories," *Formal Methods in System Design*, vol. 6, no. 2, pp. 147–189, March 1995.
4. M. D. Aagaard, R. B. Jones, T. F. Melham, J. W. O'Leary, and C.-J. H. Seger, "A methodology for large-scale hardware verification," in *Formal Methods in Computer-Aided Design: Third International Conference, FMCAD 2000: Austin, November 2000: Proceedings*, Jr. W. A. Hunt and S. D. Johnson, Eds. 2000, vol. 1954 of *Lecture Notes in Computer Science*, pp. 263–282, Springer-Verlag.
5. Mark Aagaard, Robert B. Jones, and Carl-Johan H. Seger, "Combining theorem proving and trajectory evaluation in an industrial environment," in *Design Automation Conference*, 1998, pp. 538–541.
6. M. D. Aagaard, R. B. Jones, and C.-J. H. Seger, "Formal verification using parametric representations of Boolean constraints," in *ACM/IEEE Design Automation Conference*, July 1998.
7. S. Rajan, N. Shankar, and M. K. Srivas, "An integration of model checking with automated proof checking," in *Proceedings of the 7th International Conference On Computer Aided Verification*, P. Wolper, Ed., Liege, Belgium, 1995, vol. 939, pp. 84–97, Springer Verlag.
8. J. Joyce and C.-J. Seger, "Linking BDD based symbolic evaluation to interactive theorem proving," in *ACM/IEEE Design Automation Conference*, June 1993.
9. Klaus Schneider and Dirk W. Hoffmann, "A HOL conversion for translating linear time temporal logic to omega-automata," in *Theorem Proving in Higher Order Logics*, 1999, pp. 255–272.
10. Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger, "Lifted-fl: A pragmatic implementation of combined model checking and theorem proving," in *Theorem Proving in Higher-Order Logics*. September 1999, Springer-Verlag.
11. R. Bryant, "Formal verification of memory circuits by switch-level simulation," *IEEE Transactions on ComputerAided Design of Integrated Circuits and Systems*, January 1991, Vol.10, no.1, pp. 94-102.
12. C.-J. H. Seger, "Voss — a formal hardware verification system: User's guide," Tech. Rep. TR-93-45, University of Brtish Columbia Department of Computer Science, December 1993.
13. M.J.C. Gordon, "Mechanizing programming logics in higher-order logic," in *Current Trends in Hardware Verification and Automatic Theorem Proving (Proceedings of the Workshop on Hardware Verification)*, G.M. Birtwistle and P.A. Subrahmanyam, Eds., Banff, Canada, 1988, pp. 387–439, Springer-Verlag, Berlin.
14. R. Boulton, A. Gordon, M.J.C. Gordon, J. Herbert, and J. van Tassel, "Experience with embedding hardware description languages in HOL," in *Proc. of the International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, Nijmegen, 1992, pp. 129–156, North-Holland.
15. C. M. Angelo, L. Claesen, and H. De Man, "Degrees of formality in shallow embedding hardware description languages in hol," in *Higher Order Logic Theorem Proving and its Applications: 6th International Workshop (HUG'93)*, J. J. Joyce and C.-J. H. Seger, Eds., pp. 89–100. Springer, Berlin, Heidelberg, 1994.
16. Mike Gordon, *HolBddLib, Version 2*, Computer Laboratory, University of Cambridge, March 2002.
17. Mike Gordon, *HolSatLib Documentation, Version 1.0*, Computer Laboratory, University of Cambridge, October 2001.

Formalising the translation of CTL into L_μ

Hasan Amjad

University of Cambridge Computer Laboratory, William Gates Building, 15 JJ Thomson Avenue, Cambridge CB3 0FD, UK (e-mail: Hasan.Amjad@cl.cam.ac.uk)

Abstract. The translation of the temporal logic CTL [2] into the modal μ-calculus L_μ [10] is formalised in the HOL theorem prover [8].

1 Introduction

Theorem proving and model checking are two complementary approaches to formal verification. Model checking is based on exhaustive exploration of the state space of the system under consideration. Verification is fully automatic and can provide counter-examples for debugging but suffers from the state explosion problem when dealing with complex systems. Theorem proving is based on exploring the space of correctness proofs for the system. It can handle complex formalisms but requires skilled manual guidance for verification and human insight for debugging.

HOL is based on the LCF proof assistant [7] and is written in Moscow ML. Terms are values of type `term` and can be freely constructed. Theorems are represented as values of type `thm` and can be constructed using axioms and inference rules only i.e. by proof. This reliance on minimising trusted code is often called the "fully-expansive" approach and gives a high assurance of the correctness of the results, provided the core trusted code and the underlying system (operating system, hardware etc.) are sound.

In [1] the feasibility of partially embedding a symbolic model checker for L_μ in HOL is demonstrated. This approach allows results returned from the model checker to be treated as theorems in HOL while retaining the advantages of the fully-expansive nature of HOL, without an unacceptable performance penalty.

One use of this approach would be to do property checking using logics that can be embedded into L_μ. Since L_μ is very expressive, most popular temporal logics can be embedded in it without loss of efficiency [6, 4]. A theorem returned by the model checker for an L_μ property can be translated into the corresponding theorem for whatever logic we are interested in, as long as the translation is correct. If the translation is done via a semantics-based embedding in HOL, we can use the theorem prover to prove it correct. Thus we can achieve a tight integration with established technology with little loss of efficiency, without having to write a verification tool for our new logic from scratch and with a high assurance of the correctness of our implementation. This paper provides proof-of-concept of this approach using as an example the standard embedding of the popular temporal logic CTL into L_μ. All definitions, propositions, lemmas and theorems presented here have been mechanised in HOL.

2 CTL

The temporal logic CTL is widely used to specify properties of a system in terms of the finite set AP of atomic propositions relavent to the system. We need to make the notion of "system" precise.

Definition 1. *The tuple* $(S_M, S_{0\,M}, R_M, L_M)$ *represents a Kripke structure* M *over* AP *where*

- S_M *is a finite set of states. A state is a boolean vector enumerating* AP.
- $S_{0\,M} \subseteq S_M$ *is the set of initial states.*
- R_M *is a binary transition relation on states such that* $R_M(s, s')$ *iff there is transition in* M *from* s *to* s'. R_M *is* total *i.e.* $\forall s \in S.\exists s' \in S.R(s, s')$.
- $L_M : S_M \to 2^{AP}$ *labels each state with the set of atomic propositions true in that state.*

We elide the subscript whenever the owning Kripke structure is clear from the context.

CTL allows us to describe properties of the states and paths of a *computation tree*. A computation tree is formed by unwinding (infinitely) the transitions of M, starting with the initial states. For a path π, we use π_i to denote the i^{th} state along the path, and ${}^i\pi$ to denote the prefix of π upto but not including the state π_i, and π^i to down the suffix of π starting from the state π_i.

Definition 2. *A computation path* π *starting at some state* s *in a Kripke structure* M *is well formed,* $PATHM\pi s$, *if and only if* $\pi_0 = s$, $\forall n.\pi_n \in S$ *and* $\forall n.R(\pi_n, \pi_{n+1})$.

Intuitively, a path starting with the state s is an infinite sequence of states $s_0, s_1, s_2, \ldots$ such that $s_0 = s$ and $\forall i.s_i \in S$. Paths are forced to be infinite by the totality of R.

The syntax of CTL is made out of *state formulas* which are true of states, and *path formulas* which are true of paths (where by path we just mean a sequence of states connected via transitions of the system). A well-formed CTL formula is always a state formula system. However, we need path formulas because the temporal operators talk about a state with respect to the path of the computation tree the state is on. Formally, formulas of CTL are constructed as follows:

Definition 3. *Let* AP *be the set of atomic boolean propositions. Then* CTL *is the smallest set of all state formulas such that*

- $p \in AP$ *is a state formula.*
- *If* f *and* g *are state formulas then* $\neg f$ *and* $f \wedge g$ *are state formulas.*
- *If* f *and* g *are state formulas, then* $\mathbf{X}f$, $\mathbf{F}f$, $\mathbf{G}f$, $f\mathbf{U}g$ *and* $f\mathbf{R}g$ *are path formulas.*
- *If* f *is a path formula, then* $\mathbf{A}f$ *and* $\mathbf{E}f$ *are state formulas.*

The ten compound operators thus formed can all be expressed in terms of the three operators **EX**, **EG** and **EU**. We state the following without proof (see [4]) :

Proposition 4.

- $\mathbf{AX}f = \neg\mathbf{EX}(\neg f)$
- $\mathbf{EF}f = \mathbf{E}[True\mathbf{U}f]$
- $\mathbf{AG}f = \neg\mathbf{EF}(\neg f)$
- $\mathbf{AF}f = \neg\mathbf{EG}(\neg f)$
- $\mathbf{A}[f\mathbf{U}g] \equiv \neg\mathbf{E}[\neg g\mathbf{U}(\neg f \wedge \neg g)] \wedge \neg\mathbf{EG}(\neg g)$
- $\mathbf{A}[f\mathbf{R}g] \equiv \neg\mathbf{E}[\neg f\mathbf{U}\neg g]$
- $\mathbf{E}[f\mathbf{R}g] \equiv \neg\mathbf{A}[\neg f\mathbf{U}\neg g]$

Formally, the semantics $[\![f]\!]_M$ of a CTL formula f are defined with respect to the states of the Kripke structure M. They represent the set of states of M that satisfy f. Satisfaction of f by a state s of M is denoted by $M, s \models f$. So $M, s \models f \iff s \in [\![f]\!]_M$.

Definition 5. *Given the CTL formulas f and g, atomic proposition p, a Kripke structure M and a state s of M,*

- $M, s \models p \iff p \in L(s)$
- $M, s \models f \iff M, s \not\models f$
- $M, s \models f \wedge g \iff M, s \models f \wedge M, s \models g$
- $M, s \models \mathbf{EX}f \iff \exists\pi.PATH M\pi s \wedge M, \pi_1 \models f$
- $M, s \models \mathbf{EG}f \iff \exists\pi.PATH M\pi s \wedge \forall j.M, \pi_j \models f$
- $M, s \models \mathbf{E}[f\mathbf{U}g] \iff \exists\pi.PATH M\pi s \wedge \exists k.M, \pi_k \models g \wedge \forall j.j < k \Rightarrow M, \pi_j \models f$

The following lemmas will be useful later.

Lemma 6. *For any Kripke structure M and CTL formula f,*

$$[\![\mathbf{EG}f]\!]_M = [\![f \wedge \mathbf{EX}(\mathbf{EG}f)]\!]_M$$

Proof

- $\subseteq$ *direction.* For any $s \in S$,

$$\begin{aligned}
& M, s \models \mathbf{EG}f \\
\iff\ & \exists\pi.\pi_0 = s \wedge \forall j.M, \pi_j \models f \quad \text{by 5} \\
\Rightarrow\ & M, s \models f \wedge \pi_1 \models \mathbf{EG}f \\
\iff\ & M, s \models f \wedge M, s \models \mathbf{EX}(\mathbf{EG}f) \quad \text{by 5} \\
\iff\ & M, s \models f \wedge \mathbf{EX}(\mathbf{EG}f) \quad \text{by 5}
\end{aligned}$$

and we are done by extensionality.

– $\supseteq$ *direction.* For any $s \in S$,

$$\begin{aligned} & M, s \models f \wedge \mathbf{EX}(\mathbf{EG}f) \\ \iff & M, s \models f \wedge \exists \pi.\pi_0 = s \wedge M, \pi_1 \models (\mathbf{EG}f) \quad \text{by 5} \\ \iff & \exists \pi.\pi_0 = s \wedge \forall j.M, \pi_j \models f \quad \text{by 5} \\ \iff & M, s \models \mathbf{EG}f \quad \text{by 5} \end{aligned}$$

and we are done by extensionality. □

Effectively this is saying that $\mathbf{EG}f$ is a fixed point of the function $\tau(W) = f \wedge \mathbf{EX}(W)$. We have a similar result for $\mathbf{E}[f\mathbf{U}g]$.

Lemma 7. *For any Kripke structure M and CTL formulas f and g,*

$$[\![\mathbf{E}[f\mathbf{U}g]]\!]_M = [\![g \vee (f \wedge \mathbf{EX}(\mathbf{E}[f\mathbf{U}g]))]\!]_M$$

Proof For any $s \in S$,

$$\begin{aligned} & M, s \models \mathbf{E}[f\mathbf{U}g] \\ \iff & \exists \pi.\pi_0 = s \wedge \exists k.M, \pi_k \models g \wedge \forall j.j < k \Rightarrow M, \pi_j \models f \quad \text{by 5} \end{aligned}$$

Now consider the cases $k = 0$ and $k \neq 0$. In the first case, we have $M, s \models g$. In the second case, we have $\forall j.j < k \Rightarrow M, \pi_j \models f$. But $k \neq 0$ so certainly $M, \pi_0 \models f$ i.e. $M, s \models f$. Further, $M, \pi_1 \models \mathbf{E}[f\mathbf{U}g]$ i.e. $M, s \models \mathbf{EX}(\mathbf{E}[f\mathbf{U}g])$, by 5. Since one of the two cases on k must hold, we have $M, s \models g \vee (M, s \models f \wedge M, s \models \mathbf{EX}(\mathbf{E}[f\mathbf{U}g])$ and we have the required result by 5. □

3 L_μ

Formulas of L_μ also describe properties of a system that can be represented as a state machine. As with CTL, the semantics of a formula is the set of states of the system for which the formula holds true.

The greater expressive power of L_μ requires a slightly modified version of the Kripke structure presented earlier. If AP is the set of atomic propositions relevant to our system, then a Kripke structure is defined as follows:

Definition 8. *A Kripke structure M over AP is a tuple (S, S_0, T, L) where*

- *S is a finite set of states. A state is a boolean vector enumerating AP.*
- *$S_0 \subseteq S$ is the set of initial states.*
- *T is the set of actions such that for any action $a \in T$, $a \subseteq S \times S$.*
- *$L : S \to 2^{AP}$ labels each state with the set of atomic propositions true in that state.*

Instead of a single transition relation, we now have a set of transition relations called actions. Note that the transition relations need not be total and therefore paths need not be infinite.

We now present the syntax and semantics of L_μ, essentially as given in [4]. The types of overloaded operators will be clear from the context.

Definition 9. *Let $VAR = \{Q_i | Q_i \subseteq S\}$ be the set of relational variables. Then,*

- *True and False are formulas.*
- *If $p \in AP$, then p is a formula.*
- *A relational variable is a formula.*
- *If f and g are formulas, then $\neg f$, $f \wedge g$ and $f \vee g$ are formulas.*
- *If f is a formula and $a \in T$, then $[a]f$ and $\langle a \rangle f$ are formulas.*
- *If $Q \in VAR$ and f is a formula, then $\mu Q.f$ and $\nu Q.f$ are formulas, subject to the constraint that all occurances of Q in the negated normal form of f are not negated.*

We often use the term "variable" instead of "relational variable" or "propositional variable"; the meaning should be clear from the context. In the formulas $\mu Q.f$ and $\nu Q.f$, μ and ν are considered binders on Q, and thus there is the standard notion of bound and free variables. We use $f(Q_1, Q_2, \ldots)$ to denote that $Q_1, Q_2, \ldots$ occur free in f.

Intuitively, the propositional fragment behaves as expected. In the modal fragment $[a]f$ holds of a state if f holds in all states reachable from that state by doing an a action, and $\langle a \rangle f$ holds of a state if it is possible to make an a action to a state in which f holds. We abbreviate $(s, s') \in a$ by $s \xrightarrow{a} s'$. It is often convenient to use a dot to denote an arbitrary action. In the recursive fragment, $\mu Q.f$ and $\nu Q.f$ represent the least and greatest fixpoints of the properties represented by f.

The semantics of a formula f is written $[\![f]\!]_M e$, where M is a Kripke structure and the environment $e : VAR \to 2^S$ holds the state sets corresponding to the free variables of f. By $e[Q \leftarrow W]$ we mean the environment that has $e[Q \leftarrow W]Q = W$ but is the same as e otherwise. We use $\perp$ to denote the empty environment. We now define $[\![f]\!]_M e$.

Definition 10. *The semantics of L_μ are defined recursively as follows*

- $[\![True]\!]_M e = S$ *and* $[\![False]\!]_M e = \emptyset$
- $[\![p]\!]_M e = \{s | p \in L(s)\}$
- $[\![Q]\!]_M e = e(Q)$
- $[\![\neg f]\!]_M e = S \backslash [\![f]\!]_M e$
- $[\![f \wedge g]\!]_M e = [\![f]\!]_M e \cap [\![g]\!]_M e$
- $[\![f \vee g]\!]_M e = [\![f]\!]_M e \cup [\![g]\!]_M e$
- $[\![\langle a \rangle f]\!]_M e = \{s | \exists t. s \xrightarrow{a} t \wedge t \in [\![f]\!]_M e\}$
- $[\![[a]f]\!]_M e = \{s | \forall t. s \xrightarrow{a} t \Rightarrow t \in [\![f]\!]_M e\}$
- $[\![\mu Q.f]\!]_M e$ *is the least fixpoint of the predicate transformer* $\tau : 2^S \to 2^S$ *given by* $\tau(W) = [\![f]\!]_M e[Q \leftarrow W]$

- $[\![\nu Q.f]\!]_M e$ *is the greatest fixpoint of the predicate transformer* $\tau : 2^S \to 2^S$ *given by* $\tau(W) = [\![f]\!]_M e[Q \leftarrow W]$

Environements can be given a partial ordering $\subseteq$ under componentwise subset inclusion. Now the semantics evaluate monotonically over environments [10],

Proposition 11. *For any Kripke structure* M*, environments* e *and* e'*, relational variable* Q *and well-formed* L_μ *formula* f*, and* $W \subseteq S$ *and* $W' \subseteq S$*, we have*

$$e \subseteq e' \wedge W \subseteq W' \Rightarrow [\![f(Q)]\!]_M e[Q \leftarrow W] \subseteq [\![f(Q)]\!]_M e'[Q \leftarrow W']$$

so by Tarksi's fixpoint theorem in [14], the existence of fixpoints is guaranteed. In fact, since S is finite, monotonicity implies continuity [14], which gives

Proposition 12.

$$[\![\mu Q.f]\!]_M e = \bigcup_i \tau^i(\emptyset) \text{ and } [\![\nu Q.f]\!]_M e = \bigcap_i \tau^i(M.S)$$

where $\tau^i(Q)$ is defined by $\tau^0(Q) = Q$ and $\tau^(i+1) = \tau(\tau^i(Q))$. So we can compute the fixpoints by repeatedly applying τ to the result of the previous iteration, starting with $[\![False]\!]_M e$ for least fixpoints and $[\![True]\!]_M e$ for greatest fixpoints. Since S is finite, the computation stops at some $k \leq |S|$, so that the least fixpoint is given by $\tau^k(False)$ and the greatest fixpoint by $\tau^k(True)$.

4 The Translation

The semantics for both CTL and L_μ are in terms of sets of states. This allows a purely syntactic translation scheme [4].

Definition 13. *The translation* $\mathcal{T}$ *from* CTL *to* L_μ *is defined by recursion over* CTL *formulas as follows*

- $\mathcal{T}(p \in AP) = p$
- $\mathcal{T}(\neg f) = \neg\mathcal{T}(f)$
- $\mathcal{T}(f \wedge g) = \mathcal{T}(f) \wedge \mathcal{T}(g)$
- $\mathcal{T}(\mathbf{EX}f) = \langle . \rangle \mathcal{T}(f)$
- $\mathcal{T}(\mathbf{EG}f) = \nu Q.f \wedge \langle . \rangle Q$
- $\mathcal{T}(\mathbf{E}[f\mathbf{U}g]) = \mu Q.g \vee (f \wedge \langle . \rangle Q)$

We need to prove this translation correct with respect to the semantics. Since the underlying models for CTL and L_μ are slightly different, we need to be able to translate a CTL model into an L_μ model. We overload $\mathcal{T}$ for this purpose.

Definition 14. *If* M *is a Kripke structure as given in Definition 1,* $\mathcal{T}M$ *is the Kripke structure*

$$(S_M, S_{0M}, \lambda a.R_M, L_M)$$

Theorem 15.

$$\forall M f.[\![f]\!]_M = [\![\mathcal{T}(f)]\!]_{\mathcal{T}M}\bot$$

Proof By induction on the definition of f.

- $f \equiv p$.

$$\begin{aligned} & [\![p]\!]_M \\ = & \{s | p \in L_M(s)\} \quad \text{by 5} \\ = & \{s | \mathcal{T}(p) \in L_{\mathcal{T}M}(s)\} \quad \text{by 13, 14} \\ = & [\![\mathcal{T}(p)]\!]_{\mathcal{T}M}\bot \quad \text{by 10} \end{aligned}$$

- $f \equiv \neg f'$.

$$\begin{aligned} & [\![\neg f']\!]_M \\ = & S \backslash [\![f']\!]_M \quad \text{by 5 and set theory} \\ = & S \backslash [\![\mathcal{T}(f')]\!]_{\mathcal{T}M}\bot \quad \text{by the IH} \\ = & [\![\mathcal{T}(\neg f')]\!]_{\mathcal{T}M}\bot \quad \text{by 10, 13} \end{aligned}$$

- $f \equiv f' \wedge f''$.

$$\begin{aligned} & [\![f' \wedge f'']\!]_M \\ = & [\![f']\!]_M \cap [\![f'']\!]_M \quad \text{by 5 and set theory} \\ = & [\![\mathcal{T}(f')]\!]_{\mathcal{T}M}\bot \cap [\![\mathcal{T}(f')]\!]_{\mathcal{T}M}\bot \quad \text{by the IH} \\ = & [\![\mathcal{T}(f' \wedge f'')]\!]_{\mathcal{T}M}\bot \quad \text{by 10, 13} \end{aligned}$$

- $f \equiv \mathbf{EX} f'$. For any state $s \in S_M$,

$$\begin{aligned} & s \in [\![\mathbf{EX} f']\!]_M \\ \iff & M, s \models \mathbf{EX} f' \quad \text{by definition of} [\![-]\!]_M \\ \iff & \exists \pi. PATHM\pi s \wedge M, \pi_1 \models f' \quad \text{by 5} \\ \iff & \exists \pi. PATHM\pi s \wedge \pi_1 \in [\![f']\!]_M \quad \text{by definition of} [\![-]\!]_M \end{aligned}$$

$$\begin{aligned} \iff & \exists \pi. PATHM\pi s \wedge \pi_1 \in [\![\mathcal{T}(f')]\!]_{\mathcal{T}M}\bot \quad \text{by the IH} \\ \iff & \exists \pi. R_M(s, \pi_1) \wedge \pi_1 \in [\![\mathcal{T}(f')]\!]_{\mathcal{T}M}\bot \quad \text{by 2} \\ \iff & \exists \pi. s \dot{\to} \pi_1 \wedge \pi_1 \in [\![\mathcal{T}(f')]\!]_{\mathcal{T}M}\bot \quad \text{by definition of} \to \end{aligned}$$

Now define our existential witness π by

$$\begin{aligned} \pi_0 &= s \\ \pi_(n+1) &= (n = 0)?\ s' \mid \varepsilon r. R_M(\pi_n, r) \end{aligned}$$

where ε is Hilbert's selection operator. Then simplifying and continuing,

$$\begin{aligned}
&\iff \exists s'.s \dot{\to} s' \wedge s' \in [\![\mathcal{T}(f')]\!]_{\mathcal{T}M}\bot \quad \text{by definition of}\pi \\
&\iff s \in \{s|\exists s'.s \dot{\to} s' \wedge s' \in [\![\mathcal{T}(f')]\!]_{\mathcal{T}M}\bot\} \text{ by defn of} \in \\
&\iff s \in [\![\langle . \rangle \mathcal{T}(f')]\!]_{\mathcal{T}M}\bot \quad \text{by 10} \\
&\iff s \in [\![\mathcal{T}(\mathbf{EX}f')]\!]_{\mathcal{T}M}\bot \quad \text{by 13}
\end{aligned}$$

and we have the required result by extensionality.

– $f \equiv \mathbf{EG}f'$. Define

$$\tau(W) = [\![\mathcal{T}(f) \wedge \langle . \rangle Q]\!]_{\mathcal{T}M} \bot [Q \leftarrow W]$$

and we have

- $\subseteq$ direction.

$$\begin{aligned}
&[\![\mathbf{EG}f]\!]_M \subseteq [\![\mathcal{T}(\mathbf{EG}f)]\!]_{\mathcal{T}M}\bot \\
\iff &[\![\mathbf{EG}f]\!]_M \subseteq \bigcap_n \tau^n S_{\mathcal{T}M} \quad \text{by 12,13,10,14} \\
\iff &\forall n.[\![\mathbf{EG}f]\!]_M \subseteq \tau^n S_{\mathcal{T}M} \quad \text{by set theory}
\end{aligned}$$

Then induction on n gives

  * $n \equiv 0$. Immediate by 10,14,12.
  * $n \equiv n' + 1$. Consider the "outer" IH,

$$\begin{aligned}
&[\![\mathbf{EG}f]\!]_M \subseteq \tau^{n'} S_{\mathcal{T}M} \\
\Rightarrow\ &\tau([\![\mathbf{EG}f]\!]_M) \subseteq \tau(\tau^{n'} S_{\mathcal{T}M}) \text{ by 11} \\
\iff &[\![\mathbf{EG}f]\!]_M \subseteq \tau^{n'+1} S_{\mathcal{T}M} \text{ by 6,13,10}
\end{aligned}$$

which is the required result.

- $\supseteq$ direction.

$$\begin{aligned}
&[\![\mathbf{EG}f]\!]_M \supseteq [\![\mathcal{T}(\mathbf{EG}f)]\!]_{\mathcal{T}M}\bot \\
\iff &[\![\mathbf{EG}f]\!]_M \supseteq \bigcap_n \tau^n S_{\mathcal{T}M} \quad \text{by 12,13,10,14}
\end{aligned}$$

Now consider some $s \in \bigcap_n \tau^n S_{\mathcal{T}M}$. By 12,

$$s \in \tau\big(\bigcap_n \tau^n S_{\mathcal{T}M}\big)$$

Suppose π is a path starting at s. Then by the definition of τ, $\pi_1 \in \bigcap_n \tau^n S_{\mathcal{T}M}$. We use the ε operator to pick π_1 for us (this is needed

because totality of $R.M$ only tells us that π_1 exists). By repeatedly using ε to pick the appropriate next state on the path, we can construct π such that $\forall i.\pi_i \in \bigcap_n \tau^n S_{\mathcal{T}M}$. But for any s', $s' \in \bigcap_n \tau^n S_{\mathcal{T}M} \Rightarrow M, s' \models f$ by the definition of τ and the outer IH. Thus we have that $\forall i.M, \pi_i \models f$, and we have the required result by 5.

– $f \equiv \mathbf{E}[f'\mathbf{U}f'']$. Define

$$\tau(W) = [\![\mathcal{T}(f'') \vee (\mathcal{T}(f') \wedge \langle . \rangle Q)]\!]_{\mathcal{T}M} \perp [Q \leftarrow W]$$

and we have

• $\subseteq$ direction.

$$\begin{aligned} & [\![\mathbf{E}[f'\mathbf{U}f'']]\!]_M \subseteq [\![\mathcal{T}(\mathbf{E}[f'\mathbf{U}f''])]\!]_{\mathcal{T}M}\perp \\ \iff & [\![\mathbf{E}[f'\mathbf{U}f'']]\!]_M \subseteq \bigcup_n \tau^n\emptyset \quad \text{by 12,13,10,14} \end{aligned}$$

Now consider some $s \in [\![\mathbf{E}[f'\mathbf{U}f'']]\!]_M$. Then by 5 we have,

$$\exists\pi.PATHM\pi s \wedge \exists k.M, \pi_k \models f'' \wedge \forall j.j < k \Rightarrow M, \pi_j \models f'$$

We proceed by induction on the length $|{}^k\pi|$ of ${}^k\pi$.

* $|{}^k\pi| = 0$. Note that since π is infinite, $|{}^k\pi| = k$ by the definition of ${}^k\pi$. So $k = 0$. This implies $M, s \models f''$ by 2 i.e. $s \in [\![f'']\!]_M$ and we are done by the definition of τ and the outer IH.
* $|{}^k\pi| = k' + 1$. We note again that $k = k' + 1$. Consider the path π^1. Then,

$$\begin{aligned} & M, \pi_k \models f'' \\ \iff & M, \pi^1_{k'} \models f'' \end{aligned}$$

and

$$\begin{aligned} & \forall j.j < k \Rightarrow M, \pi_j \models f' \\ \iff & \forall j.j + 1 < k \Rightarrow M, \pi_{j+1} \models f' \\ \iff & \forall j.j < k' \Rightarrow M, \pi^1_j \models f' \end{aligned}$$

So by the IH,

$$\begin{aligned} & \pi^1_0 \in \bigcup_n \tau^n\emptyset \\ \iff & \pi_1 \in \bigcup_n \tau^n\emptyset \\ \iff & s \in \tau\big(\bigcup_n \tau^n\emptyset\big) \quad \because M, s \models f' \text{ and defn of } \tau \end{aligned}$$

and we are done by the outer IH.

- $\supseteq$ direction.

$$\begin{aligned}
& [\![\mathbf{E}[f'\mathbf{U}f'']]\!]_M \supseteq [\![\mathcal{T}(\mathbf{E}[f'\mathbf{U}f''])]\!]_{\mathcal{T}M}\bot \\
\iff & [\![\mathbf{E}[f'\mathbf{U}f'']]\!]_M \supseteq \bigcup_n \tau^n\emptyset \text{by 12,13,10,14} \\
\iff & \forall n.[\![\mathbf{E}[f'\mathbf{U}f'']]\!]_M \supseteq \tau^n S_{\mathcal{T}M} \text{by set theory}
\end{aligned}$$

Then induction on n gives
  * $n \equiv 0$. Immediate by 10,14,12.
  * $n \equiv n' + 1$. Consider the outer IH,

$$\begin{aligned}
& [\![\mathbf{E}[f'\mathbf{U}f'']]\!]_M \supseteq \tau^{n'}\emptyset \\
\Rightarrow \quad & \tau([\![\mathbf{E}[f'\mathbf{U}f'']]\!]_M) \supseteq \tau(\tau^{n'}\emptyset) \text{ by 11} \\
\iff & [\![\mathbf{E}[f'\mathbf{U}f'']]\!]_M \supseteq \tau^{n'+1}\emptyset \text{ by 7,13,10}
\end{aligned}$$

which is the required result.

□

5 Concluding Remarks

Closely related work includes the HOL-Voss system [13, 9] and the integration of an L_μ symbolic model checker with PVS [12, 11]. Voss has a lazy functional language FL with BDDs as a built-in datatype. In [9] Voss was interfaced to HOL and verification using a combination of deduction and symbolic trajectory evaluation (STE) was demonstrated. The interface does not allow a tight integration however: properties verified in Voss cannot be used as theorems in HOL without invoking oracles, which may compromise the assurance of soundness provided by the fully-expansive nature of HOL. In [12] this tight integration is achieved within the PVS framework. However, the proof step invoking the model checker is atomic and thus this approach does not extend readily to the fully-expansive approach used in HOL.

Since this result has been mechanised in HOL, we can convert a CTL property to L_μ, use the L_μ property checker, and convert the resulting theorem back to a CTL property. In general, we can leverage our existing property checker to verify specifications expressed in a new logic (embeddable in L_μ) without risking unsoundness caused by an incorrect translation. This may seem trivial for CTL but the translations of other popular logics such as LTL or CTL* into L_μ are considerably more involved [5, 3] and the chance of an incorrect implementation correspondingly higher. Our fully-expansive approach towards integrating model-checking and theorem-proving removes this possibility assuming only the soundness of the HOL kernel and the operating environment.

References

1. H. Amjad. Programming a symbolic model checker in a fully expansive theorem prover. In *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science. Springer-Verlag, 2003.
2. M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.
3. E. M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design*, 10(1):47–71, 1997.
4. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.
5. M. Dam. CTL* and ECTL* as fragments of the modal mu-calculus. *Theoretical Computer Science*, 126(1):77–96, 1994.
6. E. A. Emerson and C-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *1st Annual Symposium on Logic in Computer Science*, pages 267–278. IEEE Computer Society Press, 1986.
7. M. J. C. Gordon, A. R. G. Milner, and C. P. Wadsworth. Edinburgh LCF: A mechanised logic of computation. LNCS. Springer-Verlag, 1979.
8. The HOL Proof Tool. `http://hol.sf.net`, 2003.
9. J. Joyce and C. Seger. The HOL-Voss system : Model checking inside a general-purpose theorem prover. In *Higher Order Logic Theorem Proving and its Applications*, volume 780 of *LNCS*, pages 185–198. Springer, 1993.
10. D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
11. Sam Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. Pvs: Combining specification, proof checking, and model checking. In *CAV'96: 8th International Conference on Computer Aided Verification*, pages 411–414, 1996.
12. S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking and automated proof checking. In *Proceedings of Computer Aided Verification*. Springer-Verlag, 1995.
13. C-J. H. Seger. Voss - a formal hardware verification system: User's guide. Technical report, The University of British Columbia, December 1993. UBC-TR-93-45.
14. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

Implementing Abstraction Refinement for Model Checking in HOL

Hasan Amjad

University of Cambridge Computer Laboratory, William Gates Building, 15 JJ Thomson Avenue, Cambridge CB3 0FD, UK (e-mail: Hasan.Amjad@cl.cam.ac.uk)

Abstract. Abstracting infinite or large state spaces to ones feasible for model checking has met with much success. We have implemented an abstraction framework in HOL, on top of a deep-embedded model checker. We present the implementation, highlighting the role of HOL.

1 Introduction

Model checking and theorem proving are two complementary approaches to formal verification. Model checking models the system as a state machine and desired properties of the system are expressed as temporal logic formulae that are true in the desired states of the system. Verification is fully automatic and can provide counter-examples for debugging but suffers from the state explosion problem when dealing with complex systems. Theorem proving models the system as a collection of definitions and desired properties are proved by formal derivations based on these definitions. It can handle complex systems but requires skilled manual guidance for verification and human insight for debugging.

An increasing amount of attention has thus been focused on combining these two approaches (see [19] for a survey). In this paper we demonstrate an approach to embedding a model checker in a theorem prover. The expectation is that this will ease combination of state-based and definitional models and the respective property checking techniques. Model checkers are typically written in tightly optimised C with an emphasis on performance. Theorem provers typically are not. Preliminary benchmarking shows that the loss in performance using our approach is within acceptable bounds.

Since our emphasis is on security (in the sense of soundness not being compromised), we have chosen the HOL theorem prover [14] for our task. HOL is based on the HOL logic [12] which is an extension of Church's simple theory of types [5], and is written in Moscow ML. Terms (of ML type `term`) in the logic can be freely constructed. Theorems (of ML type `thm`) can be constructed using the core axioms and inference rules only, i.e. by proof. This reliance on a very small trusted core is often named the "fully-expansive" approach and gives a high assurance of security.

Symbolic model checking [15] is a popular model checking technique. Sets of states are represented by the BDDs [3] of their characteristic functions. This representation is compact and provides an efficient[1] way to test set equality and do

[1] The problem is NP-complete. So this efficiency is of heuristic value only.

image computations. This is useful because evaluating temporal logic formulae almost always requires a fixed point computation that relies on image computations to compute the next approximation to the fixed point and a set equality test to determine termination. Most of the work is done by the underlying BDD engine. Current model checking techniques are typically unable to verify real-world examples due to the large number of states involved. Abstraction [9, 17, 13, 8] is considered a promising approach in handling this problem.

By representing primitive BDD operations as inference rules added to the core of the HOL theorem prover, we can model the execution of a model checker for a given property as a formal derivation tree rooted at the required property [1]. These inference rules are hooked to a high performance BDD engine [4] external to the theorem prover. Thus the loss of performance is low, and the security of the theorem prover is compromised only to the extent that the BDD engine or the BDD inference rules may be unsound. Since we do almost everything within HOL and use only the most primitive BDD operations, we expect a higher assurance of security than from an implementation that is entirely in C.

We now build upon this embedded model checker embedded in HOL by adding an abstraction framework. As before, the fully-expansive approach is used and all steps in the computation are justified by HOL proofs. This retains the high assurance of soundness, and also allows us to use the decision procedures and simplifiers of HOL without loss of compositionality.

2 Abstraction Refinement in HOL

Abstraction techniques reduce the number of states of a system so that it is more amenable to model checking. This is typically done using functional abstraction [8] or Galois connections [13]. Here we consider the functional abstraction approach supplemented with a refinement framework [6]. A functional abstraction typically computes an abstraction function h that is a surjection from the set of states of the system under consideration to the domain of abstract states.

We need to make the notion of "system" precise. For our purposes, the system is represented by a Kripke structure. If AP is the set of atomic propositions relevant to the system we wish to model, then a Kripke structure over AP is defined as follows:

Definition 1. *A Kripke structure M over AP is a tuple (S, S_0, T, L) where*

- *S is a finite set of states.*
- *$S_0 \subseteq S$ is the set of initial states.*
- *T is the set of actions (or transitions or program letters) such that for any action $a \in T$, $a \subseteq S \times S$.*
- *$L : S \to 2^{AP}$ labels each state with the set of atomic propositions true in that state.*

Each $p \in AP$ is a proposition constructed from the variables v of some finite domain D_v and the constants and operators over that domain. Let $V =$

$\bigcup_{p \in AP} freevars(p)$ and $n = |V|$. Thus the set $S = D_{v_0} \times D_{v_1} \times \ldots D_{v_{n-1}}$. A state $s \in S$ in M is thus a tuple over V. We write $s \vDash p$ if p is true when the variables of p are assigned the corresponding values from the state s.

If there is a transition a taking state s to state s', we write $R_a(s, s')$. The notation $R(s, s')$ indicates that there is some transition from s to s'. The value of a variable v in the next state is denoted by v'. For each transition a, the transition relation R_a is given by $\bigwedge_{i=0}^{n-1} v_i' = f_i(s)$ for synchronous and $\bigvee_{i=0}^{n-1} v_i' = f_i(s)$ for asynchronous systems, where the f_i are *next state formulae* over the variables of s that determine what the value of v_i should be in the next state. We write $M \vDash \phi$ if the temporal property ϕ constructed over the p_i holds in all states of M.

The abstraction function $h : S \to \hat{S}$ is a surjection to the set $\hat{S}$ of abstract states. In general we can use h to compute the abstract model $\hat{M} = (\hat{S}, \hat{S}_0, \hat{T}, \hat{L})$ as follows[2]:

1. $\hat{S}$ is the set of abstract states $\hat{s}$ where the $\hat{s}$ are partitions of S
2. $\hat{S_0}(\hat{s}) = \exists s.h(s) \iff \hat{s} \wedge s \in S_0$
3. $\hat{R}(\hat{s_1}, \hat{s_2}) \iff \exists s_1 s_2.h(s_1) = \hat{s_1} \wedge h(s_2) = \hat{s_2} \wedge R(s_1, s_2)$
4. $\hat{L}(\hat{s}) = \bigcup_{h(s)=\hat{s}} L(s)$

This is called *existential abstraction*.

Theorem 2. *For any temporal property ϕ, $\hat{M} \vDash \phi \Rightarrow M \vDash \phi$*

Proof Sketch The abstraction adds extra behaviours to $\hat{M}$ that were not present in M. However it does not take away any behaviour i.e. it computes an over approximation. Thus if a property holds in the abstract it will hold in the concrete. If a property fails in the abstract it may be because of the spurious behaviour. □

If we limit ourselves to universal properties, then if a property fails in the abstract system we can generate a counterexample trace in the abstract system and attempt to find a corresponding concrete trace. If one exists than the property is false in the concrete system and the verification fails. Otherwise the abstraction is too coarse and we refine it by splitting some abstract state into smaller sets of concrete states. We then recheck the property. This is continued until either the property is verified or a concrete counterexample found. We are guaranteed termination because each refinement is strict (i.e. the sets resulting from splitting an abstract state are non-empty) and eventually we will end up with the concrete system that cannot be further refined. Figure 2 shows the overall framework.

2.1 Generating the initial abstraction

The first step is to generate the initial abstraction. First, we partition AP (and hence V) into sets that do not have a variable in common. This induces a partitioning on S. Then for each partition we group together into one abstract state all concrete states that have transitions to the same concrete states.

[2] NOTE: Sets in HOL are identified with higher-order predicates i.e. $x \in P \iff Px$. We will use the predicate notation where it is convenient.

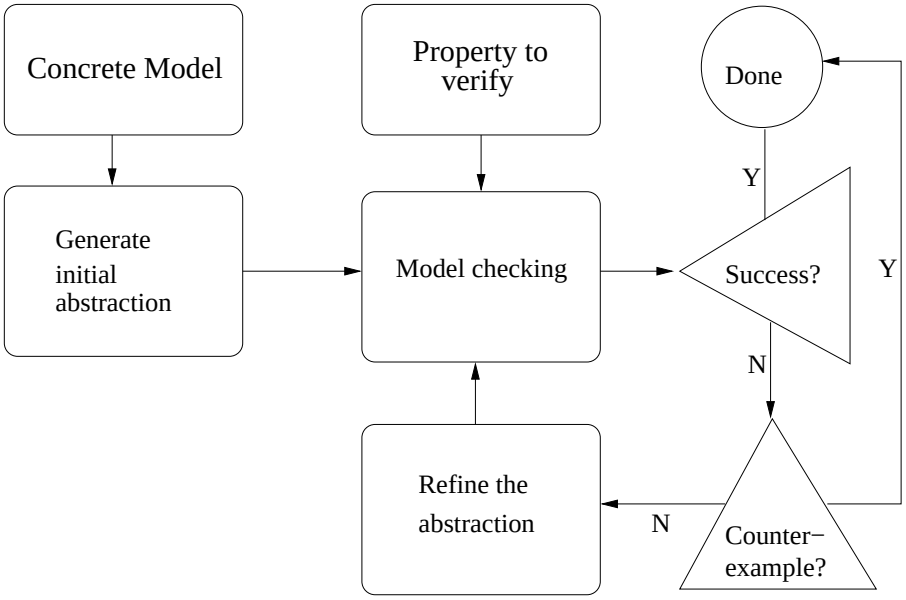


Fig. 1. Overview of Abstraction Refinement Framework

More precisely,

1. Let $(v_0, v_1, \ldots, v_n)$ be any state s_i.
2. Let I be the set of initial states and R be the transition relation given by $\bigwedge_i v_i' = f_i(s)$. Let F be the set of all next-state formulae f_i.
3. Let $f_i \equiv_f f_j \iff vars(f_i) \cap vars(f_j) \neq \emptyset$.
4. Let FC_i be the partitions $PART$ of F induced by $\equiv_f$.
5. Let $VC_i = \bigcup_{f \in FC_i} vars(f)$. Let $D_{VC_i} = \prod_{v_j \in VC_i} D_{v_j}$.
6. Let $h_i : D_{VC_i} \rightarrow \hat{D_{v_i}}$ where $\hat{D_{v_i}}$ is part of the abstract domain, be defined by $h_i(s_j) = h_i(s_k) \iff \forall f \in FC_i . s_j \vDash f \iff s_k \vDash f$
7. Then the abstraction function is $h = (h_0, \ldots, h_{|PART|-1})$.

The FC_i are called formula clusters, and the VC_i are called variable clusters. The domains D_{VC_i} represent the partitions of S. Note that the set of initial states S_0 is not used in the construction: $\hat{S_0}$ is simply the set $h(S_0)$ where h has been extended to sets in the obvious way. We can now use existential abstraction to construct $\hat{M}$.

2.2 Counterexample Detection

If the verification fails, the model checker implemented in [1] has the ability to generate a counterexample trace. A counterexample is a sequence of states starting with an initial state and following transitions to a state violating the property

being verified. Our technique for detecting whether a concrete counterexample exists is that presented in [7] and is best illustrated by an example.

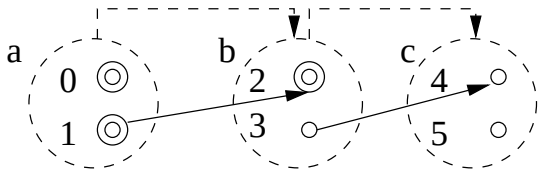


Fig. 2. Counterexample Detection Example

Figure 2.2 shows a system with $S = \{0, 1, 2, 3, 4, 5\}$. The abstract states are $\{a, b, c\}$ and the dashed circles indicate the concrete states they contain. Solid arrows represent transitions in the concrete system and dashed arrows represent transitions in the abstract system. The initial states are 0 and 1 and concentric-circles represent states reachable from the initial states. Note that the abstraction introduces extra behaviour by making states 4 and 5 reachable in the abstract system by making c reachable.

Suppose we check for a property P that holds in $\{0, 1, 2, 3\}$ but not in $\{4, 5\}$. We will get an abstract counterexample trace $\langle a, b, c\rangle$.

In general, given an abstract counterexample $\langle \hat{s_0}, \hat{s_1}, \ldots, \hat{s_k}\rangle$, we attempt to find a concrete counterexample $\langle s_0, s_1, \ldots, s_k\rangle$. To determine whether such a concrete trace exists, we try to find a satisfying assignment for the formula

$$S_0(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigwedge_{i=0}^{k} h(s_i) = \hat{s}_i$$

using a SAT-solver.

If a concrete trace is found, the verification has failed and we are done. Otherwise we set $k = 0$ and attempt to find the longest prefix of $\langle \hat{s_0}, \ldots, \hat{s_k}\rangle$ for which there is a concrete trace by running the SAT tool on the formula above for increasing values of k.

In our example, there is no concrete counterexample and the longest prefix is $\langle a, b\rangle$. This information is then used to refine the state b into smaller abstract states.

2.3 Refining the Abstraction

We would like to find the coarsest possible refinement i.e. the fewest possible splits of the abstract state. This is an intractable problem. Our technique for refinement employs BDDs to get good results.

Consider Fig. 2.3 representing the same system as Fig. 2.2. We know that we need to refine abstract state b. Since the spurious behaviour is created by an

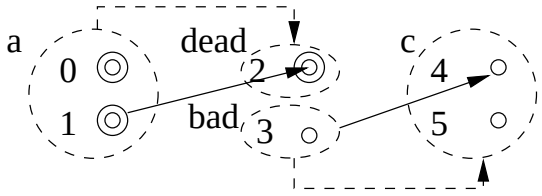


Fig. 3. Abstraction Refinement Example

unreachable state 3 in b having a transition to 4, we need to split all such states from the reachable states (in this case just $\{2\}$).

To do this, we compute the state sets

$$bad = \mathbf{EX}c = \{3\}$$

and

$$dead = b \backslash bad = \{2\}$$

where **EXf** computes all states such that there is a transition to a set in which the property represented by f is satisfied. In our case the property c yields precisely the set $\{4, 5\}$. This computation is done using standard BDD methods.

Intuitively the *dead* states are reachable "dead-ends" and the *bad* states are the ones causing the trouble by contributing to the creation of spurious behaviour. We can now replace the abstract state b in the abstraction by the abstract states *dead* and *bad* and repeat the procedure until the property is verified or a counterexample is found. In this case the abstract c is no longer reachable and this P is verified.

3 Implementation Issues

Figure 3 gives an overview of the implementation. It should be noted that the system is fully automatic and relies on fully-expansive proof at every step of the way.

The only exception to this are the dotted lines indicating the use of an external BDD engine. However, we use an LCF-style interface [11] to this engine which gives a higher assurance of soundness than integrating it as a one-shot proof rule as has been done in [16, 2].

The SAT engine we used is also external to HOL. However, checking the satisfiability of an assignment is in general much easier than finding such an assignment. Thus we use the SAT engine to obtain an assignment but check its validity by proof in HOL. Thus the use of the SAT engine does not risk introducing unsoundness in the system.

The ACTL to μ-calculus translation is not a requirement. However, recall that only universal properties can be used in this framework. The μ-calculus is a fairly non-intuitive logic and it is hard to manually check that a μ-calculus

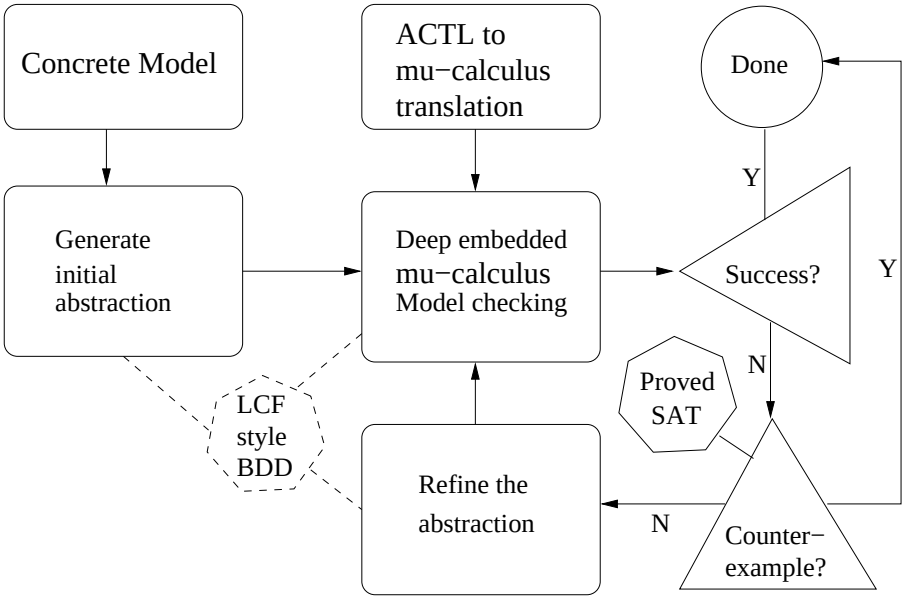


Fig. 4. Overview of Implementation in HOL

property is indeed universal. Thus we also accept properties in the more intuitive logic ACTL, which is a the universal fragment of CTL. Thus universality is enforced automatically because the HOL translation from CTL to the μ-calculus is done by proof based on the semantics of the two logics.

Note that step 6 of initial abstraction generation is in effect inducing equivalence classes of concrete states over each partition D_{VC_i} of S. When constructing a BDD representation of h to assist with the construction of $\hat{M}$ it is easier to compute these equivalence classes directly. However, the standard BDD methods for finding equivalence classed work by detecting strongly connected components (SCCs) in the graph representation of the model. SCC detection requires a total transition relation which is usual verifying properties in CTL. This is not guaranteed in our more general case with the μ-calculus. Thus we use the following algorithm for partitioning a given D_{VC_i} with respect to satisfiability.

The term $h_i(s_j) = h_i(s_k)$ can be considered as a relation $R'(s_j, s_k)$ on states. This relation induces the partitions we wish to compute. However R' can also be considered a transition relation on D_{VC_i}, with there being a transition between two states precisely when h_i agrees on them as defined by step 6 of section 2.1. Define the modality **EP** as the temporal inverse of **EX** i.e. **EP** f computes states such that there is a transition from states satisfying f to these states. Now we compute as follows:

1. Let $X = D_{VC_i}$. Let $P = []$.
2. If $X = \emptyset$ return P.

3. Let s be an arbitrary state in X (found using the BDD engines satisfiability finder).
4. Set $S = X, T = R'$ and use the model checker to compute $Y = (\mu Q.s \vee \mathbf{EX}q) \vee (\mu Q.s \vee \mathbf{EP}q)$. Then Y is the set of all states reachable from s plus all states from which s is reachable.
5. Let $P = Y :: P$
6. Let $X = X \backslash Y$ and repeat from step 2.

At the end of this the list P will contain the required partitions.

4 Related Work and Conclusion

Abstraction refinement for model checking in the context of theorem proving has also been done by [18, 2]. In [18] the abstraction framework is based on Galois connections and the refinement is done by adding the failed predicates from the previous proof attempt to get a richer abstract domain. Since the system is implemented as an atomic proof rule, access to the procedures and simplifiers in PVS itself for the purposes of the system cannot be done within the encompassing derivation tree. This inhibits a fully-expansive implementation of the system. It also restricts compositionality because for instance the system is unable to return a counterexample trace upon failure thus any counterexample guided predicate discovery system (e.g. [10]) cannot be used in this context.

In [2], a proof system for the μ-calculus relies on proof rules being executed by various decision procedures than can be plugged in according to need. This gives the framework great versatility in the choice of tools to be used to attack a given problem. Again, each proof rule is justified by an atomic call to a decision procedure and thus does not extend readily to a fully expansive approach.

Our approach is flexible in that any proof rule of HOL at any level of abstraction can be called upon at any time during the execution of the procedure. This enables us to provide the an entire run of the procedure as a derivation tree that can be plugged into any other proof. This high level of integration guarantees compositionality and makes the framework extensible. Thus we are able to use a BDD engine, a SAT engine and various HOL procedures in the same framework.

At the same time, the execution is fully expansive. All steps are accompanied by the application of a HOL proof rule. Thus we have a high assurance of soundness.

However, having to do fully expansive proof for the equivalent of fast BDD operations necessarily involves a performance penalty. To a great extent, we can ameliorate this by manipulating the higher-order equivalents of the propositional terms being manipulated by the BDD and SAT engines. Benchmarking for the current framework is pending some code optimisations. Benchmarking for the core model checker showed a performance penalty of about 20 percent as compared to the system with all proof machinery turned off [1]. This is acceptable in most situations, though the tradeoff with respect to increased assurance of soundness would need to be considered on a case-by-case basis.

In the current system the $p \in AP$ are restricted to propositional expressions. We hope to extend this to include Presburger formulae and – trading off some automation – full arithmetic and real numbers, leveraging the facilities in HOL for deciding these. Other future directions include support for other temporal logics and for other automatic abstraction and refinement techniques.

References

1. H. Amjad. Programming a symbolic model checker in a fully expansive theorem prover. In *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science. Springer-Verlag, 2003.
2. S. Berezin. *Model Checking and Theorem Proving: a Unified Framework.* PhD thesis, Carnegie Mellon University School of Computer Science, 2002.
3. R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
4. The BuDDy ROBDD Package. `http://www.itu.dk/research/buddy`, 2002.
5. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.
6. E. Clarke, O. Grumberg, S. Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *"Proc. of Conference on Computer-Aided Verification*, LNCS, pages 154–169. Springer, 2000.
7. E. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine learning techniques. In *Proc. of Conference on Computer-Aided Verification (CAV'02)*, LNCS, 2002.
8. Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
9. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM Press, 1977.
10. Satyaki Das and David L. Dill. Counter-example based predicate discovery in predicate abstraction. In *Formal Methods in Computer-Aided Design*, November 2002.
11. M. J. C. Gordon. Programming combinations of deduction and bdd-based symbolic calculation. *LMS Journal of Computation and Mathematics*, August 2002.
12. M. J. C. Gordon and T. F. Melham. *Introduction to HOL (A theorem-proving environment for higher order logic).* Cambridge University Press, 1993.
13. S. Graf and H. Saidi. Construction of abstract state graphs with pvs. In *Proceedings of Computer Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
14. The HOL Proof Tool. `http://hol.sf.net`, 2003.
15. K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.
16. S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking and automated proof checking. In *Proceedings of Computer Aided Verification.* Springer-Verlag, 1995.
17. H. Saidi. Model checking guided abstraction and analysis. In *Proceedings of the 7th International Static Analysis Symposium*, 2000.

18. H. Saidi and N. Shankar. Abstract and model check while you prove. In *Proceedings of the 11th International Conference on Computer-Aided Verification (CAV99)*, volume 1633 of *LNCS*. Springer, 1999.
19. T. E. Uribe. Combinations of model checking and theorem proving. In *Proceedings of the Third Intl. Workshop on Frontiers of Combining Systems*, volume 1794 of *LNCS*, pages 151–170. Springer-Verlag, March 2000.

Author Index