21st International Conference on Theorem Proving in Higher Order Logics

Ottane Ait Mohamed, César Muñoz and Sofiene Tahar

Technical Report
August, 2008

(2008-1-Ait Mohamed)
Preface

This technical report is the Emerging Trends proceedings of the 21th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2008), which was held during 18-21 August 2008 in Montreal, Canada. TPHOLs covers all aspects of theorem proving in higher order logics as well as related topics in theorem proving and verification. In keeping with longstanding tradition, the Emerging Trends track of TPHOLs 2008 offered a venue for the presentation of work in progress, where researchers invited discussion by means of a brief introductory talk and then discussed their work at a poster session.

July 2008

Otmene Ait Mohamed
César Muñoz
Sofiene Tahar
# Table of Contents

Using Isabelle/HOL for Static Program Verification in JML4 .......................... 1  
  *Patrice Chalin, Perry R. James, George Karabotsos*

Theorem Proving based Framework for Verification of Group Key Protocols .......................... 9  
  *Amjad Gawanmeh*

Probabilistic Analysis using Theorem Proving ................................................... 21  
  *Osman Hasan*

Checking the Satisfiability of XML-Specifications .............................................. 33  
  *Harald Hiss*

Reasoning with Powerdomains in Isabelle/HOLCF .............................................. 45  
  *Brian Huffman*

Shallow Dependency Pairs ................................................................. 57  
  *Alexander Krauss*

A reflexive formalization of a SAT solver in Coq ............................................ 64  
  *Stéphane Lescuyer, Sylvain Conchon*

Formally Specifying and Proving Operational Aspects of Forensic Lucid in Isabelle .......................... 76  
  *Serguei Mokhov, Joey Paquet*

Verification of LISP Interpreters .......................................................... 99  
  *Magnus O. Myreen*

Focus on Isabelle: From Specification to Verification .................................... 104  
  *Maria Spichkova*

A Separation Logic Framework in HOL ......................................................... 116  
  *Thomas Tuerk*

Implementing Secure Broadcast Ambients in Isabelle using Nominal Logic .......................... 123  
  *Ayesha Yasmeen, Elsa Gunter*
Using Isabelle/HOL for Static Program Verification in JML4

Patrice Chalin, Perry R. James, George Karabotsos

Dependable Software Research Group,
Dept. of Computer Science and Software Engineering,
Concordia University, Montréal, Canada
{chalin, perry, g_karab}@dsrg.org

Abstract. JML4 is an Integrated Verification Environment (IVE) for JML-annotated Java that builds upon Eclipse’s support for Java. Two forms of static verification are provided: Extended Static Checking (ESC) and Full Static Program Verification (FSPV). In this paper we explain how use is made of Isabelle/HOL in support of both ESC and FSPV.

1 Introduction

The Java Modeling Language (JML) is the most popular Behavioral Interface Specification Language (BISL) for Java. JML is recognized by a dozen tools and used by over two dozen institutions for teaching and/or research, mainly in the context of program verification [8]. Tools exist to support the full range of verification from Runtime Assertion Checking (RAC) to Full Static Program Verification (FSPV) with Extended Static Checking (ESC) in between [3].

In this paper we describe JML4, a next generation Eclipse-based Integrated development and Verification Environment (IVE) for Java and JML [5]. In particular, our focus is on how JML4 makes use of Isabelle/HOL both as an Automated Theorem Prover (ATP) and an Interactive Theorem Prover (ITP) in support of ESC and FSPV, respectively. JML4 can output lemmas (corresponding to unproven verification conditions) in theory files whose proof scripts can be completed offline to allow future invocations of ESC to successfully prove the correctness of methods. Simpl is an Isabelle/HOL theory that provides syntax and semantics of an imperative language. As an alternative to ESC, JML4’s FSPV outputs a theory file—one lemma per method—encoded in Simpl.

In the following section, we provide background material on Isabelle, Simpl (a logic built atop Isabelle/HOL), and JML. Section 3 presents the functionality provided by JML4, with an emphasis on its two forms support for Static Verification (SV), both of which make use of Isabelle/HOL. ESC makes use of it as an Automatic Theorem Prover (ATP), while FSPV TG makes use of Simpl. Section 4 gives an overview of the architecture of the SV subsystems. A discussion of the FSPV Theory Generator (TG) is given in Section 5. Conclusion and future work are presented in Section 6.

2 Background

2.1 Isabelle

Isabelle is an interactive theorem prover. Its development is lead by Laurence C. Paulson of the University of Cambridge and Tobias Nipkow of Technische Universität München. In fact Isabelle is more than just a theorem prover: it is a theorem proving framework. It provides enough proving machinery to define new logics. This machinery includes Isabelle’s meta-logic (Isabelle/Pure), the classical reasoner, and the simplifier. Additionally, existing logics can be extended, thus defining new ones. Isabelle/HOL is just one of these logics defined on top of Isabelle/Pure. As the name suggests, Isabelle/HOL is a realization of High Order Logic for Isabelle. Isabelle/HOL is the most complete of all object logics written for Isabelle so far. This reason among others is why Isabelle/HOL has served as the basis for a number of additional logics. Some of these include the logic for Computable Functions

---

1 Contrary to some other authors, we interpret the term “full verification” to imply interactive verification.
(Isabelle/HOLCF), and logics for sequential imperative programs with Hoare semantics defined such as Bali and Simpl [12]. Of interest to us are the Hoare-based logics—in particular we are interested in Simpl.

2.2 Simpl

Simpl is one of the most recent theories that deals with the logic of programming to be published in Isabelle’s Archive of Formal Proofs [12]. It was published just this past February and was developed by Norbert W. Schirmer of Saarland University. Simpl is a theory for a sequential imperative programming language. Within its boundaries, syntax and big- and small-step operational semantics are defined. A set of Hoare rules for both partial and total correctness is defined, along with their proofs of soundness and completeness. Simpl provides for a shallow embedding of the target language. In other words Simpl is independent of a particular syntax. This simplifies the logic immensely by concentrating on semantics rather than syntactic manipulations. Nonetheless, it is expressive enough for many language constructs that exist in modern programming languages. Some of these include global and local variables, exceptions, abnormal termination, breaks out of loops, procedures, and expressions with side-effects.

A Simpl theory is structured around a state and a Simpl program. The state takes the form of a `hoarestate` which is used to denote the list of variables used in a Simpl program. Hoarestates are build on top of Isabelle’s Statespaces, which can be thought of as records with multiple inheritance that make use of locales. The program can be written within a definition, or as is more often done, within a lemma. A program is similar to a Hoare triple where by a precondition is followed by the program definition, which is finally followed by a postcondition. Tactics are defined that automate verification, these are the vcg-step tactic, which performs verification step by step, and vcg, which performs many steps automatically.

2.3 JML

The Java Modeling Language (JML) is a BISL that was designed for use by Java developers having only modest mathematical training [10]. JML’s syntax is similar to that of Java, and its annotations are given in specially formatted comments. Leavens, Baker, and Ruby state that JML is “more expressive … than Eiffel and easier to use than VDM and Larch,” as it combines the best features of these other approaches. Compared to the Larch BISLs, JML is simpler and easier to understand since its syntax is similar to Java’s [9].

JML can document both the interface provided by Java code and its behavior, so it is well suited to documenting detailed designs. Interface specifications include annotations on declarations with extended type information, such as that used by universal types or the non-null type system. The behavioral specifications often specify pre- and postconditions that state properties that should hold when the Java code is executed as well as which—and under what conditions—exceptions may be thrown by the code. In keeping with its goal to allow for incremental adoption, JML allows specifications to range from being detailed and complete to being as little as a single clause giving a single property [7].

Several tools have been developed that process JML-annotated code. These tools provide support for Runtime Assertion Checking (RAC), Extended Static Checking (ESC), Full Static Program Verification (FSPV), automatic discovery of invariants, automated unit testing, and documentation generation [3].

3 JML4

3.1 Overview

A wide variety of tools have been developed that support JML, but many of these are showing their age. For example, neither the JML Compiler, which instruments Java bytecode with Runtime Assertion Checks, nor the ESC/Java2, an Extended Static Checker, provides support for features introduced in Java 5, such as generics.

JML4 is a next-generation tools framework that is based on the Eclipse Java Development Toolkit (JDT). It’s goal is to reduce the effort needed to develop new JML tools by providing access to a JML-decorated AST, without requiring the JML community to maintain the underlying Java compiler. While the JDT is large—approximately 1 MLOC for 5000 files—and the learning curve is steep (partly due to lack of documentation) we nonetheless chose to take the plunge and began prototyping JML4 in 2006.
In our first feature set, JML4 enhanced Eclipse 3.3 with scanning and parsing of nullity modifiers, enforcement of JML’s non-null type system (both statically and at runtime) and the ability to read and make use of the extensive JML API library specifications. This architecturally significant subset of features was chosen so as to exercise some of the basic capabilities that any JML extension to Eclipse would need to support. These include

- recognizing and processing JML syntax inside specially marked comments, both in *.java files as well as *.jml files;
- storing JML-specific nodes in an extended AST hierarchy,
- statically enforcing a modified type system, and
- generating runtime assertion checking (RAC) code.

The chosen subset of features was also seen as useful in its own right [4], somewhat independent of other JML features. In particular, the capabilities formed a natural extension to the existing embryonic Eclipse support for nullity analysis.

We mention in passing that in parallel with our work on next generation components we have integrated the two main first-generation JML tools: ESC/Java2 and the JML RAC. Hence, at a minimum, JML users actively developing with first generation tools will be able to continue to do so, but now within the more hospitable environment offered by Eclipse.

3.2 Current capabilities

With respect to the next generation components proper, at the time of writing, JML4 parser support is nearing what is called JML Level 1, which includes the most frequently used core JML constructs [11, §2.9]. Basic support for RAC (e.g., inline assertions and simple contracts) is available, but our research group is leading development in static verification components—details of our work to date are given in the next section. Yet others are exploring the integration of new tools for JML. We are hopeful that next-generation components fully processing JML Level 1 specifications will be ready by the fall of 2008.
Besides work on the JML4 infrastructure, our research group has been focusing its efforts on the development of a new component called the JML Static Verifier (JML SV). This new component offers the basic capabilities of ESC and FSPV.

The ESC component of JML4, referred to as ESC4, is a ground-up rewrite of ESC which is based on Barnett and Leino’s innovative and improved approach to a weakest precondition semantics for ESC [2]. Our FSPV tool, called the FSPV Theory Generator (TG), is like the JML LOOP compiler [13] in that it generates theories containing lemmas whose proof establish the correctness of the compilation unit in question. The FSPV TG currently generates theories written in the Hoare Logic of SIMPL—an Isabelle/HOL based theory designed for the verification of sequential imperative programs [12]. Lemmas are expressed as Hoare triples. To prove the correctness of such lemmas, a user can interactively explore their proof using the Eclipse version of Proof General (PG) [1]—see Figure 1.

3.3 Static Verification (SV): ground-up designs using latest techniques

Besides work on the JML4 infrastructure, our research group has been focusing its efforts on the development of a new component called the JML Static Verifier (JML SV). This new component offers the basic capabilities of ESC and FSPV.

The ESC component of JML4, referred to as ESC4, is a ground-up rewrite of ESC which is based on Barnett and Leino’s innovative and improved approach to a weakest precondition semantics for ESC [2]. Our FSPV tool, called the FSPV Theory Generator (TG), is like the JML LOOP compiler [13] in that it generates theories containing lemmas whose proof establish the correctness of the compilation unit in question. The FSPV TG currently generates theories written in the Hoare Logic of SIMPL—an Isabelle/HOL based theory designed for the verification of sequential imperative programs [12]. Lemmas are expressed as Hoare triples. To prove the correctness of such lemmas, a user can interactively explore their proof using the Eclipse version of Proof General (PG) [1]—see Figure 1.

3.4 Innovative SV features

In addition to supporting ESC and FSPV, the JML SV currently supports the following features, most of which are novel either in the context of verification tools in general or JML tools in particular:

- Multi Automated Theorem Prover (ATP) support including:
– First-order ATPs: Simplify and CVC3.
– Isabelle/HOL, which, we have found can be used quite effectively as an ATP.

- A technique we call 2D Verification Condition (VC) cascading where VCs that are unprovable are broken down into sub-VCs (giving us one axis of this 2D technique) with proofs attempted for each sub-VC using each of the supported ATPs (second axis).
- VC proof status caching. VCs are self-contained context independent lemmas (because the lemma hypotheses embed the context) and hence they are ideal candidates for proof status caching. I.e., the JML SV keeps track of proven VCs and reuses the proof status on subsequent passes, matching textually VCs and hence avoiding expensive re-verification.
- Offline User Assisted (OUA) ESC, which we explain next.

By definition, ESC is fully automatic [6] whereas FSPV requires interaction with the developer. OUA ESC offers a compromise: a user is given an opportunity to provide (offline) proofs of sub-VCs which ESC4 is unable to prove automatically. Currently, ESC4 writes unprovable lemmas to an Isabelle/HOL theory file (one per compilation unit). The user can then interactively prove the lemmas using Proof General. Once this is done, ESC4 will make use of the proof script on subsequent invocations. We have found OUA ESC to be quite useful because ESC4 is generally able to automatically prove most sub-VCs, hence only asking the user to prove the ones beyond ATP abilities greatly reduces the proof burden on users.

Figure 2 sketches the relationship between the effort required to make use of each of the JML SV verification techniques vs. the level of completeness that can be achieved. Notice how ESC4, while requiring no more effort to use than its predecessor ESC/Java2, is able to achieve a higher level of completeness. This is because ESC4 makes use of multiple prover back-ends including the first order provers Simplify and CVC3 as well as Isabelle/HOL. As was mentioned earlier, Isabelle/HOL can be used quite effectively as an automated theorem prover; in fact, Isabelle is able to (automatically) prove the validity of assertions that are beyond the capabilities of the first order provers—e.g., assertions making use of quantifiers. Another example of a method which JML SV can prove correct using Isabelle/HOL as an ATP is Cube.java given in Figure 1 (the reason ESC4 shows that it is unable to prove the loop invariants is because we disabled use of Isabelle/HOL as an ATP for illustrative purposes—to contrast with what can be proven using Cube.thy).

In summary, the latest features added to JML4 make it the first IVE for a mainstream programming language to support the full range of verification technologies (from RAC to FSPV). Its innovative features make it easier to achieve complete verification of JML annotated Java code and this more quickly: preliminary results show that ESC4 will be at least 5 times faster than ESC/Java2. Furthermore, features like proof caching, and other forms of VC proof optimization, offer a further 50% decrease in verification time. Of course, until JML SV supports the full JML language, these results are to be taken as preliminary, but we believe that they are indicative of the kinds of efficiency improvements that can be expected.

In the next section, we explore the architecture of JML4 in general, and the JML SV in particular, allowing us to see how the features described above have been realized.

4 Architecture of JML Static Verifier (SV)

In this section we present an architectural overview of JML4’s Static Verifier (rather than the full compiler or other aspects of the IDE).

As was explained above, the JML SV supports two main kinds of verification: extended static checking—both the normal kind and Offline User Assisted (OUA) ESC—and full static program verification. These are realized by the subcomponents named ESC4 and the FSPV Theory Generator (TG), respectively. A diagram illustrating dataflow for the JML SV is given in Figure 3. The input to the JML SV is a fully resolved AST for the compilation unit actively being processed.
Initiated on user request, separate from the normal compilation process, the FSPV TG generates Isabelle/HOL theory files for the given compilation unit (CU). One theory file is generated per CU. The theory files can then be manipulated by the user using Proof General.

When activated (via compiler preferences), ESC4 functionality kicks in during the normal compilation process following standard static analysis. The ESC phases are the standard ones [6], though the approach used by ESC4 is new in the context of JML tooling: it is a realization of the Barnett and Leino approach [2] used in Spec# in which the input AST is translated into VCs by using a novel form of guarded-command program as an intermediate representation. The Proof Coordinator decides on the strategy to use: e.g. single prover, cascaded VC proofs or OUA ESC. In the latter case, Isabelle theory files are consulted when sub-VCs are unprovable and a user-supplied proof exists. Unfortunately, the detailed design and full details of the behavior of the JML SV are beyond the scope of this paper.

5 Translation to Simpl

The latest addition to the JML4 tooling is the Full Static Program Verification (FSPV) Theory Generator (TG) component. This component is responsible for translating JML annotated Java classes into Simpl theories. Then a user can interactively prove the Simpl theory—i.e. prove the program’s correctness. With the advent of Proof General’s plug-in for eclipse, a user can seamlessly and effortlessly switch between development and verification within a single environment, thus facilitating the use of formal verification techniques on the JML-annotated Java programs.

The FSPV TG has been recently incorporated into the JML tooling—as such it still provides a very limited support for JML and Java. Nevertheless it has provided us with the proof of concept for its incorporation and usage into our tooling.

The TG supports a limited but functional subset of JML annotations. This includes basic lightweight contracts with requires and ensures clauses as well as loop annotations with maintaining and decreasing clauses. TG supports a small but functional subset of Java that includes

1. basic types such as int, boolean, and char,
2. local variables and method parameters,
3. expressions with side-effects, such as assignment and binary expressions with most of the common arithmetic and relational operators, and
4. statements that include if statements, while loops, and local variables with or without initialization.
As can be seen in Figure 4, a single theory is created per compilation unit (CU). The theory is named after the CU’s file name. For each method in the CU, a single Simpl hoarestate and program is generated. The names of both are formed from the method’s name and parameter types to disambiguate overloaded methods. The hoarestate also contains _vars as a suffix. The hoarestate collects and defines the parameters and local variables used in the method. The Simpl program is defined within the confines of an Isabelle lemma statement. The Simpl pre- and postcondition are created by conjoining and translating the method’s requires and ensures clauses, respectively. The Simpl program is a translation of the JML and Java expressions and statements to their corresponding ones in Simpl. Finally applications of the vcg and auto tactics are added to guide the proof of the lemma. It may very well be that these tactics are not appropriate for discharging this lemma—in such case the user should use the appropriate ones in their place. A more complex example of a CU and its corresponding, automatically generated theory can be seen in Figure 1.

Although Simpl supports the definition of local variables we choose not to use them in this initial integration of the TG in JML4. The reason for this was to keep the integration as simple as possible so to facilitate the evaluation of the overall tooling. A better integration is planned in the near future, for more on this we refer you to section 6.

### 6 Conclusion and Future Work

JML4 is an Integrated Verification Environment (IVE) for JML-annotated Java that is based on Eclipse. It has been built taking into consideration the lessons learned during the development of the first-generation JML tools, as well as taking advantage of advancements of tools for other, related languages. JML4’s Static Verification makes available Extended Static Checking that uses multiple automatic theorem provers, including Isabelle/HOL. Its Full Static Program Verification Theory Generator (FSPV TG) relies on Proof General as an interface to Isabelle to prove the correctness of methods as a whole.

There is much work yet to be done. The most pressing is completing the compiler front-end to support all or most of JML. Once this is done, it will be necessary to propagate support for JML constructs into the RAC, ESC and FSPV components.

### References


Abstract. The correctness of group key protocols in communication systems remains a great challenge because of dynamic characteristics of group key construction as we deal with open number of group members. In this paper, we present a combination of three different theorem-proving methods to verify security properties for group oriented protocols. In the first method, rank theorems for forward properties are established based on a set of generic formal specification requirements for group key management and distribution protocols. Rank theorems imply the validity of the security property to be proved, and are deduced from a set of rank functions we define over the protocol. In the second, we provide a sound and complete inference system to detect attacks in group key management protocols. The inference system provides an elegant and natural proof strategy for such protocols compared to existing approaches. Finally, in the third method, we use an event-B first-order proving system to provide invariant checking for group key secrecy property. In our framework, we applied each method on a different group protocol from the literature illustrating the features of each method.

1 Introduction

Security protocols are used to establish secure channels between communicating systems. Great care needs to be taken in developing and implementing robust protocols. The complexity of security-protocol interactions can hide security weaknesses that normal analysis methods cannot reveal, mainly because they involve precise interactions in order to achieve the required security services, therefore, it is very important to verify that the protocol operations are not vulnerable to attacks. Besides, networks handle more and more tasks in a potentially hostile environment. Therefore, cryptographic protocols should take more responsibilities in order to capture these new requirements.

There are different kinds of environments that protocols must interoperate with, besides, networks handle more and more tasks in a potentially hostile environment. Therefore, cryptographic protocols should take more responsibilities in order to capture these new requirements. This of course, makes both modeling and verification more difficult. It also requires the search for new modeling and verification approaches for cryptographic protocols.

Group key protocols allow group members to exchange or establish keys to encrypt and authenticate messages within the group. At the same time, individuals outside of the group cannot eavesdrop on group communication or inject messages. In fact, group key management protocols need security retention in the case of dynamic member actions, such as leaving the group for an existing member, or joining the group for a new member. It should be also guaranteed that all authorized members are able to access the group, at the same time unauthorized ones are unable to have this access.

Therefore, security properties that are well defined in normal two-party protocols have different meanings and different interpretations in group key distribution protocols. An example of such properties is secrecy, which deals with the fact that secret data should remain secret and not compromised. However, for group key distribution protocols, this property has a further dimension since there are long-term secret keys, short-term secret keys, in addition to present, future, and past keys; where a principal who just joined the group and learned the present key should not be able to have enough information to deduce any previous keys, or similarly a principal who just left the group should not have enough information to deduct any future keys. Therefore, systems designed for two-party protocols may not be able to model a group protocol, or its intended security properties because such tools require an abstraction to a group of fixed size to be made before the automated analysis takes place. This can eliminate chances of finding attacks on these protocols.

In this paper, we describe a framework for the verification of group key management protocols. The framework is based on three approaches: a rank theorems based approach, a rank functions based inference system, and and
event-B first-order logic theorem proving approach. In the first two methods we adapted the idea of rank functions introduced by Schneider et al. [12, 21] in order to be able to verify security properties for group oriented protocols. In our method, we first define a set of sound rank functions that satisfy specific requirements, and prove the correctness of every rank function with these requirement. Then, we use this set of rank functions in two directions, in the first to define rank theorems that guarantee the correctness of the security property. The implementation of the rank theorems methods, from our experiences, requires a considerable amount of effort and time in theorem proving. To enhance the performance of this technique we combine the rank function into the protocol events. This resulted in the second method, a rank functions based inference system, which requires the same level of higher-order logics to reason about messages and events. The inference system is composed of a set of inference rules over rank functions, where, every rule can be applied in order to generate new knowledge and assign new ranks to these generated messages. A special rule called Attack is defined to represent the bottom of the system, and, when executed, illustrates an attack in the protocol. We formally prove the soundness and the completeness of the rank theorem and the inference system for a sound rank functions.

Implementing the inference system in higher-order logic theorem proving required a lot of effort and time, in addition, verifying properties is achieved interactively with the theorem proving tool because of the decidability problem on higher-order logic. In the third method, which is complimentary to the above two, we define a formal link between event-B semantic and security protocols, the use of event-B first-order logic prover to verify secrecy property as an event-B invariant. This allows us to avoid the user interaction with the theorem proving tool, and reduce the time required to verify the security property. This is not a straightforward task, and should be based on a correct semantical link between the two models, and finally the application of a group protocol (that has join/leave events) making use of first-order logic automatic prover, while abstracting some features of the protocol.

The above methods were applied on different protocols from the literature. First we implemented the rank theorem proof environment on the Enclaves protocol from SRI [11] in order to verify related forward secrecy using the PVS (Prototype Verification System) theorem prover [18]. Then we implemented the verification of a generic Group Diffie-Hellman (GDH) protocol [24, 10] in PVS based on the proposed inference system approach. Finally, we applied the event-B based approach on a tree based Group Diffie-Hellman (TGDH) protocol and used invariant checking to verify the correctness of key construction. Protocol events and traces are modeled as event-B operations and messages as sets. Then, group secrecy property was defined and verified as an event-B invariant in tool Click’n’Prove [3].

The rest of the paper is organized as follows. Section 2 discusses related work to ours. In Section 3, we overview the general framework and preliminary definitions and notations we use. In Section 4, we define the concept of rank theorems, and we show how to define theorems for forward and backward secrecy properties. In Section 5, we introduce the details of our rank functions based inference system. In Section 6, we present the event-B based approach. Finally, Section 7 concludes the paper with future work hints.

2 Related Work

In this section we discuss work related to ours in the literature to the best of our knowledge.

Pereira and Quisquater [19] proposed a systematic approach to analyze protocol suites extending the Diffie-Hellman key-exchange scheme to a group setting. They pointed out several unpublished attacks against the main security properties claimed in the definition of these protocols. The method provided is essentially manual and applicable only on Group Diffie-Hellman (GDH) protocols. In a more recent work, Pereira and Quisquater [20] provided a systematic way to derive an attack against any Authenticated GDH-type (A-GDH) protocol with at least four participants and exhibit protocols with two and three participants. They provided a generic insecurity results concerning authentication protocols. In their work, the authors did not attempt to address group related properties such as forward and backward secrecy.

In another related work, Steel et al. [23] modeled a group key protocol by posing inductive conjectures about the trace of messages exchanged in order to investigate novel properties of the protocol, such as tolerance to disruption, and whether it results in an agreement on a single key. The method, however, is applicable on limited groups of two or three members only. Recently, Truderung [26] presented a formalism, called selecting theories, which extends the standard non-recursive term rewriting model and allows participants to compare and store arbitrary messages. This formalism can model recursive protocols, where participants, in each protocol step, are able to send a number of
messages unbounded w.r.t. size of the protocol. This modeling, however, cannot be applied on non–recursive protocols such as GDH or the Enclaves. In addition, the model provided is not readable and very complex to construct. There are many other efforts in the literature that deal with the formal analysis for GDH style protocols, for more details see [14].

Dutertre and Schneider [12] used an embedding of CSP (Communication Sequential Process) in PVS in order to verify the authentication property of the Needham-Shroeder public key protocol. They proposed the idea of rank functions in order to enable CSP verification of the Needham Shroeder protocol. Later, Ryan and Schneider [21, 22] used the idea of rank functions for the verification of CSP. The work did not present a method that can be applied on security properties in other classes of protocols, specifically, group key protocols. In fact, the method, as is, may not be applied on secrecy property for group key management protocols. Even though rank functions were introduced and used by Schneider et al. [12, 22, 21, 8, 9] in different directions, in this paper, we use their definition in order to precisely define a set of sound rank functions and prove their correctness. We then propose a rank functions based inference system for the verification of group key distribution protocols and prove the soundness and completeness of the system using the defined set of rank functions.

Layouni et al. [17] used a combination of model checking, theorem proving, and a Random Oracle Model to verify authentication property, safety and liveness properties such as proper agreement, and robustness and unpredictability properties, respectively. The verified protocol is a complex protocol developed for group key agreement under multiple leaders scheme, and is called the Enclaves protocol [11]. This example shows how difficult it is to verify and analyze this class of protocols. While the authors achieved a promising success in verifying a complex protocol such as Enclaves, they failed to accomplish the formal proof of the three components in a single formalism.

Events-based verification of security protocols was used by Crazzolara and Winskel [6, 7] for mappings between process algebra, Petri nets, strand spaces and inductive models. The authors established precise relationships between the Petri nets semantics and transition semantics, strand spaces, inductive rules, and trace languages and event structures. They show how event-based models can be structured in a compositional way and so used to give a formal semantics to security protocols which support proofs of the correctness of these protocols. They demonstrated the usefulness of their Petri nets semantics in deriving proof principles for security protocols and apply them to prove an authentication property. The method is applicable to authentication property insufficient for capturing forward and backward secrecy properties.

Stouls and Potet [25] proposed a method to automatically enforce an abstract security policy on a network. They used the B refinement process to build a formal link between concrete and abstract terms, which is dynamically computed from the environment data. They applied their method on a case study modeling a network monitor. A different approach to achieve a similar objective was proposed in [4], where the authors addressed the proof-based development of system models satisfying a security policy. They used OrBAC [1] models to express the security policies in order to state permissions and prohibitions on actions. An abstract B model [2] is derived from the OrBAC specification of the security policy and then the model is refined to introduce properties that can be expressed in OrBAC. The refinement guarantees that the resulting B model satisfies the security policy.

In this paper we address security property for group oriented protocols, taking into account features such as the concept of group secrecy and dynamic group events. In addition, we consider events that are specific for group protocols that were not treated by the B method based work of Butler [5]. The verification is tackled using theorem proving in higher-order and first-order logic models.

3 Verification Methodology

Figure 1 displays a general view of the proposed verification methodology. Three major techniques are used: in the first one, we use rank theorems to define the required security property based on the concept of rank functions. Rank theorem are implemented in PVS higher-order logic theorem prover. In the second approach, we provide a sound and complete inference system to detect attacks in group key management protocols. The inference system is also implemented in PVS. Finally, abstract our group model to fit it to the first-order logic theorem proving in event-B. We define a well-formed formal link between the group protocol model and the event-B counterpart model. Based on this link, we propose a solution to verify the required security properties, in particular secrecy properties, using the
event-B invariant checking tool Click’n’Prove. First we present our formal model and the notations that will be used throughout this paper.

![Diagram of Formal Protocol Model]

**Fig. 1. Verification Methodology**

- **M**: set of all possible messages (messages space).
- **P**: a honest principal who is willing to communicate.
- **P**: set of knowledge of member **P**, \( P \subseteq M \).
- **S**: secret messages space, the set of all secret messages, \( S \subseteq M \). These are the messages we want to keep hidden from the intruder. They are defined by the protocol.
- **I**: a dishonest member. We assume that the intruder is a dishonest member who is trying to find an attack in the protocol by using his unlimited resources and computational power. However, we state normal assumptions about the intruder such as being able to encrypt or decrypt a message only if he knows the appropriate key, or the ability to block or read any message in the system.
- **E**: set of all events, or dynamic operations, i.e., join, leave, merge, and split. An event is a term from the message space to the message space, \( E : M \rightarrow M \). It represents an action the user can perform in order to obtain extra information and update his own set of knowledge.
- **T**: set of all possible traces, where a trace of events is the execution of the sequence of these events. We use \( \tau \in T \), such that \( \tau : E \times M \rightarrow M, m \in M \), then we write \( m = \tau(E,M) \) to say that a message \( m \) is generated by the trace \( \tau \) by executing the vector of events \( E \) on the set of messages \( M \), we also write \( \tau(E,M) \rightsquigarrow m \) to represent a predict formula that evaluates to true if and only if \( m = \tau(E,M) \).
- **K_0**: set of initial knowledge of the intruder, where \( K_0 \subseteq M \). The initial knowledge of the intruder is basically the information he/she can collect before executing the protocol events. This information is usually public and known, so there are no secret information that is in the intruders initial set of knowledge. In other words \( \forall m \in M : m \in S \Rightarrow m \notin K_0 \).
- **K**: set of knowledge of the intruder. The intruder updates this knowledge by executing events. The intruder starts with the initial set of knowledge and the set of events, then, by executing a sequence of events, he/she updates this set. \( K_0 \subseteq K \) and \( K \subseteq M \).
- **attack**: we define attack w.r.t. confidentiality as the ability of the intruder to have a message in the set of secret messages in his own set of knowledge, \( \text{attack} \equiv m \in K \) and \( m \in S \). The notion of attack can be seen differently depending on the nature of the security property under investigation.

Traces are defined since they will be used to prove the soundness of the inference system. \( t \) represents a single execution of one event or inference rule that updates the intruder set of knowledge. For a given events \( e_1, e_2, ..., e_n \in E \), we use \( M \) to represent the messages generated by executing each event: \( m_1 = e_1(M_0), m_2 = e_2(M), ..., m_n = \)
where $M_0 \in K_0$ is a set of messages in the initial set of knowledge of the intruder, and $M^p \in K_0$ is a vector of $p$ messages in the updated set of knowledge of the intruder. Now we can define $t_1, t_2, ..., t_n$ as follows:

$t_1 : m_1 = e_1(K_0), \rho(m_1) = c_1, K_1 = K_0 \cup \{m_1\}$
$t_2 : m_2 = e_2(K_1), \rho(m_2) = c_2, K_2 = K_1 \cup \{m_2\}$
$t_i : m_i = e_i(K_i), \rho(m_i) = c_i, K_i = K_{i-1} \cup \{m_i\}$

... 
$t_n : m_n = e_n(K_n), \rho(m_n) = c_n, K = K \cup \{m_n\}$

We define a trace $T_n \in T$ as the sequence of executing $t_1, t_2, ..., t_n$ in order.

$T_n : t_1, t_2, ..., t_n, K = K_0 \cup \{m_1, m_2, ..., m_n\}; \rho(m_n) = c_n$. We say that $m_n = T_n(E, M)$ which means that the trace $T_n$ generates the message $m_n$ of rank $c_n$.

Rank functions were first introduced in [21]. For the purpose of establishing the proof that a specific fact will not be available to the intruder, we assign a value or rank to each fact, such that, facts that can be generated by the system have positive rank, and facts that cannot be obtained by the intruder cannot have positive rank. The definition of the rank function is as follows:

**Definition 1.** A rank function $\rho$ is a function $\rho: M \rightarrow Z$ that maps the set of all messages into integers.

It is necessary to verify that protocol participants cannot generate non-positive ranks. The appropriate rank function we choose to apply on the protocol should be sound. We define a set of rank functions with a number of requirements, which will be used to prove the correctness of the rank function.

**Definition 2.** A rank function is sound if it satisfies the following condition: applying an event on the intruder’s initial set of knowledge will not generate a zero rank:

$$\forall m \in K_0, e \in E, \rho(m) > 0 \text{ and } \rho(e(m)) > 0.$$ 

This intuitively means that the rank function will not generate or add any inconsistency to the group model. Therefore, it will guarantee that zero ranks can only be generated as a result of problems in the protocol.

**Definition 3.** A rank function is initially sound if it satisfies these three requirements:

1. $\forall m \in M, \rho(m) >= 0$, there are no negative ranks generated by the system.
2. $\forall m \in K_0, \rho(m) > 0$, intruder initial knowledge must be of positive rank.
3. $\forall m \in S, \rho(m) = 0$, all secret messages must have a zero rank.

**Definition 4.** Two events are invertible if each one is the inverse of the other.

$e_2(e_1(m_1)) = m_1$, where $e_1, e_2 \in E$, and $m_1 \in M$.

**Definition 5.** A rank function is invertible for all invertible events of inference rules. $e_2(e_1(m_1)) = m_1 \Rightarrow \rho(e_2(e_1(m_1))) = \rho(m_1)$, where $e_1, e_2 \in E$, and $m_1, m_2 \in M$, and any user of the system cannot apply an invertible event unless he/she is able to apply the inverse.

**Definition 6.** A rank function is bounded if $\rho(m) - 1 \leq \rho(e(m)) \leq \rho(m) + 1$, where $e \in E$, and $m \in M$.

**Theorem 1.** Rank Function Soundness

A rank function is sound, if it is initially sound, invertible and bounded.

The theorem states that a rank function with the above specifications is consistent. It ensures that the zero rank cannot be generated by the initial knowledge of the intruder, or by the definition of the rank function for the events. In other words, applying each single event separately on the set of intruders initial knowledge will not generate a zero rank, simply because a secret is not revealed to the intruder. Formally, $\forall m \in K_0, \rho(m) > 0$ and $\forall m \in K_0, e \in E, \rho(e(m)) > 0$. We use $\mathbb{R}$ to represent the set of all sound rank functions. More details about the theorem and its proof can be found in [14]. In the next sections we talk in details about each of the three methods of the framework.
4 Rank Theorems

We briefly present the steps of our rank theorems based verification methodology as described Figure 2. The first step consists of providing a formal model and precise definition for group protocols properties and events. This will help eliminating the gap between the informal protocol specification and the formal model. It will also provide a well defined protocol specification that can be directly integrated into the verification methodology. In the second step, we define map functions between the set of facts and the set of integers. The set of facts include protocol events, protocol execution traces and the security property. Then we define rank theorems that provide conditions satisfied by a given rank function in order to conclude that the security property satisfies its protocol model. We define for every security property a theorem that implies the validity of the property with respect to the protocol. Rank theorems imply the correctness of the security property it models. In order to prove the correctness of a specific property, we need to prove that its corresponding rank theorem is correct with respect to the protocol model. In the final step of our approach, we implement the rank theorems in PVS and establish their proof of correctness.

In order to make sure of the soundness of our approach, we have to show the formal link between the constructed rank theorem and the formal model of the property. This is carried out by proving that the correctness of the rank theorem implies the correctness of the security property. This way, we can argue that verifying the property at the implementation level guarantees the correctness of the property in the model.

The rank function should obey specific rules in order to be sound. First there are no negative ranks generated by the system. It is necessary to verify that each participant cannot introduce anything of non-positive rank to the system. In other words, the intruder initial knowledge must be of positive rank, and only facts of positive ranks can be generated from sets of facts of positive rank. All messages that are supposed to be secret and unknown to the intruder are mapped to zero rank. When executing an event, the rank of the generated message is a bounded function of the rank of the parameters of the event.

We define a property \( \phi \) for a given group protocol \( G \). This property states that a dishonest user \( I \) cannot execute a trace in \( T \) in order to discover a secret in \( S \), and is formally modeled as follows: \( \phi = \forall \tau \in T, \ (En, Mp) \rightarrow m \Rightarrow m \not\in S \). If this property is correct for the protocol \( G \) then we can write \( G \models \phi \). This is a general secrecy property that will be used to define and proof the rank theorem. The target security property to be verified, i.e., forward secrecy,
will be concretely defined later in this section. Now, we define and prove a general rank theorem for this property as follows:

**Theorem** \( \forall m \in K, \rho(m) > 0 \Rightarrow G_t \models \phi \), where \( m = \tau(E, M) \) and \( \tau \in T \).

This means that for all traces \( \tau \in T \), a dishonest principal \( I \) can execute on a group protocol \( G_t \). We say that the protocol satisfies a security property \( \phi \), \( G_t \models \phi \), if the protocol can maintain a positive rank for the messages that can be generated by the intruder.

Now we define our forward secrecy property, \( \phi \), based on the formal specifications model presented previously as follows: \( \phi = \forall m \in S, I \in G_t \Rightarrow \neg \exists \tau \in T : \tau(E, M) \rightsquigarrow m \). We use the above rank theorem in order to define a rank theorem for the forward secrecy property \( \phi \) (and similarly for backward secrecy). However, we should define the rank function \( \rho_{\phi} \) that maps the set of all messages to the appropriate ranks. \( \rho_{\phi} \) is defined as follows, where \( \forall i \in \mathbb{Z} : t + i \geq 0 \) and \( t - i \geq 0 \).

\[
\rho_{\phi}(m) = \begin{cases} 
0, & \text{if } m \in S \lor m = K_{G,t-i} \\
1, & \text{if } m \in K_0 \lor m = K_{G,t} \lor (m = K_{G,t+i} \land I \in G_{t-i})
\end{cases}
\]

This means that for the validity of forward secrecy, we give rank zero to all messages in the set of secret messages \( S \), such as secrets shared between users and servers, and all groups keys that were generated before the current group key. However, for the keys generated after the assumed dishonest user joined the group are mapped to a positive rank because they are in his/her initial set of knowledge. Now we can write the above theorem for forward secrecy property as follows:

**Theorem** \( \forall m \in K, \rho_{\phi}(m) > 0 \Rightarrow G_t \models \phi \), where \( m = \tau(E, M) \) and \( \tau \in T \).

The advantages of introducing such theorems, is that, first, it is protocol independent, which means that we can apply it on different protocols as well as on the same protocols at different levels of abstraction. Second, it is implementation independent, which gives more freedom to verification tool choice without any modification on the previous steps of our methodology.

This approach was illustrated on forward secrecy property for the Enclaves protocol. For further details see [15]

### 5 Rank Functions based Inference System

Our inference system consists of a set of inference rules. Every rule represents an event in the protocol. Rules have a precondition that has to be satisfied before they are applied. We define the pair \( \langle m, c \rangle \) to represent a message \( m \) and its rank \( c \). A special rule, \textit{Attack}, is defined with a precondition, such that, when executed by the intruder, it indicates the occurrence of an attack by reaching the bottom of the system \( \bot \). The intruder, by executing these rules on the set of knowledge \( K \), generates new knowledge with new ranks and updates his/her set. For this system to work, we assume the fairness of executing these rules, i.e., the intruder will not keep using the rule \textit{compose} forever, but other rules will have their chance to be executed, specially the rule \textit{Attack}. The set of rules in the inference system are shown below.
Rule 1: Encryption: 
\[ K \cup \{ \langle m, \rho_1 \rangle, \langle k, \rho_2 \rangle \} \cup \{ \langle \{ m \}_k, \rho_1 + 1 \rangle \} \]
where \( \{ m \}_k = \text{Encr}(m, k) \)

Rule 2: Decryption: 
\[ K \cup \{ \langle \text{Decr}(m, k), \rho_1 \rangle, \langle k, \rho_2 \rangle \} \cup \{ \langle \{ m \}_k, \rho_1 - 1 \rangle \} \]
where \( m = \text{Decr}(\{ m \}_k, k) \)

Rule 3: Compose: 
\[ K \cup \{ \langle m_1, \rho_1 \rangle, \langle m_2, \rho_2 \rangle \} \cup \{ \langle \text{Comp}(m_1, m_2), \min(\rho_1, \rho_2) \rangle \} \]

Rule 4: Decompose: 
\[ K \cup \{ \langle \text{Comp}(m_1, m_2), \rho_1 \rangle \} \]

Rule 5: Expo: 
\[ K \cup \{ \langle \text{expo}(m_1, m_2), \rho_1 \rangle \} \]

Rule 6: Attack: 
\[ K \cup \{ \langle m, 0 \rangle \} \]

For this inference system, we use the event Encr(m, k) to represent a message m that is encrypted w.r.t. a symmetric encryption algorithm with the key k. The event Decr(Encr(m, k), k) represents decrypting a message that is already encrypted, where the same key used for the encryption is to be used for the decryption event. These two events are invertible, therefore \( m = \text{Decr}(\text{Encr}(m, k), k) \). The event Comp(m1, m2) represents two composed messages by concatenation. The function min gives the minimum rank from two given ranks.

In the following we define theorems for the soundness and completeness of the approach. The first theorem states that if we can find a message in the set of knowledge of the intruder that has the rank zero, or equivalently, if the rule attack of the inference system is applied, then there is an attack in the system. We assume fairness in applying the inference rules.

**Theorem 2. Soundness**

Let \( P \) be a security protocol, let \( \rho \) be a sound rank function, and let \( K_0 \) be the set of the initial knowledge of the intruder. Then, the protocol \( P \) has an attack if the inference rule attack can be applied in a fair inference system.

\[ \exists m \in K : \rho(m) = 0 \text{ and } \rho \in \mathbb{R} \Rightarrow \exists \text{attack in } P. \]

The following corollary is deduced from the above theorem and states that if there is no attack in the system, then a sound rank function will be greater than zero. It can be viewed as the complementary case of the above theorem.

**Corollary 1. Absence of Attack**

Assuming the same conditions as Theorem 2, if the protocol \( P \) has no attack, then the rule attack will never be applied in a fair inference system.

\[ \neg \text{attack in } P \Rightarrow \forall m \in K : \rho(m) > 0. \]

The second theorem states that if there is no message in the set of knowledge of the intruder that has the rank zero, or equivalently, if the rule attack of the inference system can never be applied, assuming fairness of the strategy application of inference rules, then there is no attack in the system.

**Theorem 3. Completeness**

Assuming the same conditions as Theorem 2, if the rule attack cannot be applied in a fair inference system, then the protocol \( P \) has no attack.

\[ \forall m \in K : \rho(m) > 0, \rho \in \mathbb{R} \Rightarrow \neg \text{attack in } P. \]

**Corollary 2. Detecting Attacks**

Assuming the same conditions as Theorem 2, if the rule attack can be applied in a fair inference system, then the protocol \( P \) has an attack.

\[ \exists m \in K : \rho(m) = 0 \text{ and } \rho \in \mathbb{R} \Rightarrow \exists \text{attack in } P. \]
This corollary states that when the rank function evaluates to zero, then there exists an attack in the protocol. Theorems 2 and 3 and Corollaries 1 and 2 provide the formal link between the protocol model and the implementation model in PVS. The proofs of these theorems can be found in [14].

The inference system we defined can prove that an attack exists in the protocol using the Theorem 2. However, the limitation of the Soundness Theorem comes in the type of implementation that will be used. In case we want to prove that there is no attack in the protocol, the inference system can diverge in case we apply the Theorem 3. The inference system may terminate with a result, depending on the nature of the protocol and the strategies used while conducting theorem proving.

We still can reason about absence of attacks in protocols using Theorem 3. An indirect proof can be generated by proving that the strategy is fair and the application of our inference system diverges. This indirect proof can be achieved by generating partial proofs that affirm that the inference system will diverge when the applied strategy is fair. We believe that in order to apply this approach, we have to provide an implementation for the inference system that allows partial proofs based on divergence. This later issue will be considered for further study.

In order to illustrate this method, we consider the Group Diffie-Hellman (GDH) protocol [24], which is a basic group key management protocol widely studied in the literature. In the first part, we showed how to manually detect the attack in a step by step application of the inference system. Then we use the PVS theorem prover in order to implement the inference system and apply it on the protocol.

We applied our approach on the example protocol in the existence of an active adversary. The case of a passive adversary is more restricted than an active one. In addition, it has been shown that a protocol that is secure in the passive setting can be considered secure in the active case [16]. Therefore, we believe it is adequate to handle the active adversary case for the case study, and consider the passive adversary case as restricted special case which may be considered for further investigations [14].

6 Event-B Semantics based Verification Methodology

It is a practical solution to verify a security property using model checking tools, when applicable. However, it is inconvenient because of two reasons: the state space explosion problem of model checking, and the limited expressiveness of proposition logic based-tools. Treating the problem at the first-order logic level requires applying a valid abstraction on the protocol in order to fit to the proving system. This abstraction should be based on a correct semantical link between the protocol model and the target model. We tackle this problem by using event-B as the target first-order logic model benefiting from the automation and the expressiveness of the logic and the availability of supporting tools.

In order to model a group protocol in event-B first-order logic, the semantics of the event-B language should be formally related to the protocol model. We define well-formed conditions to guarantee that the event-B invariant is equivalent to the security property in the group protocol model. These conditions are particular to the group protocol model, and are essential to establish the equivalent event-B model. We show how an event-B model can be structured from group protocols model and then used to give a formal semantics to protocols which support proofs of their correctness. More precisely, we give a map from our protocol model to the event-B language. The event-B language allows the definition of invariant properties and provides an automatic model-checking based proof.

The proposed verification methodology consists of a number of steps as shown in Figure 3. In the first step, the group key protocol is specified formally using the model proposed in [14] in order to obtain precise protocol specifications. In addition, the secrecy property expected to be checked by the system is described informally. In the second step, the obtained specification is translated into event-B specification using mapping relations presented in Figure 4. From this mapping we obtain an event-B model that captures the features of the group protocol mode. Next, the secrecy property $\phi$ is specified as an invariant of the resulting event-B model $I$. Messages can be defined as a set with an enumeration of all possible secret and known messages. The intruder initial knowledge, $K_0$, is directly defined as variable or set in the event-B initialization list. Secret messages are defined similarly. Protocol initial constraints, such as $K_0 \subseteq M$ and $S \subseteq M$, are defined as properties that will be included in the invariant. Protocol join or leave events are defined as event-B operations that update the intruder’s knowledge and the set of secret messages, including the new generated key. Finally, the property is checked from the obtained global system specification using the event-B invariant checking tool Click’n’Prove.
Protocol events and execution traces are mapped into event-B events, messages generation conditions are mapped into events guards, and messages sets are used to generate event-B model constants properties. The initial knowledge is defined as event-B initializations, messages are mapped directly into sets, and finally the secrecy property is defined as an invariant for the event-B model. The generation of the target event-B model requires treating three parts: the static part which includes initializations and the constant properties of the protocol, the dynamic part that represents events of the protocol, and finally, enriching the resulting model with invariants describing the required secrecy properties.

The event-B semantics is close to the protocol model semantics. This relationship is demonstrated by establishing a well-formed link between the semantics of both models. To achieve this link, we are interested in showing that if the invariant $I$ holds for event-B machine $M$, then the safety property $\phi$ must hold for the group protocol model $G$. Formally, $(M \models I) \Rightarrow (G \models \phi)$. In terms of equivalence between the two models, we can say that a protocol model $G$ is equivalent to an event-B model $M$, with regards to the security property, if the property $\phi$ holds in the model $G$, and the invariant $I$ holds in the model $M$. To illustrate this equivalence, we need to show that $I \Rightarrow \phi$. Therefore, it is enough to show that the invariant $I$, with regards to $M$, implies the safety property $\phi$, with regard to $G$. The full details of the semantical link can be found in [13].

Under certain conditions, we guarantee that when the invariant holds in event-B model, the security property definition holds for the group protocol model. These predicates should be considered carefully when providing the event-B implementation. Properties that can be expressed as invariants are verified using the translation process and event-B tool. Other properties could be verified directly with model checking. This is illustrated in Theorem 4 shown below.
Theorem 4. Secrecy Soundness. A group protocol, $G$, satisfies its secrecy property, $\phi$, if there is an equivalent event-B model, $M$ ($G \equiv M$), that satisfies an event-B invariant, $I$, that implies the property $\phi$. Let ($G \equiv M$), ($M \models I$), and ($I \Rightarrow \phi$) be correct lemmas, then
\[(G \equiv M) \land (M \models I) \land (I \Rightarrow \phi) \Rightarrow (G \models \phi).\]

This method was applied on a Tree-based Group Diffie-Hellman protocol that generates a key in a distrusted group. We show how the conditions defined for the correctness of the above model can be concretely applied on a real protocol. The intended secrecy property, along with its conditions, are efficiently defined and checked as event-B invariant [13].

7 Conclusion

The correctness of group key protocols in communication systems remains a great challenge because of the sensitivity of the services provided. In this paper, we illustrated the need for a verification methodology for a class of protocols that deal with group key distribution.

A framework for modeling and verification of group key management protocols that combine three approaches is proposed. We defined a formal model for group key protocols based on a set of generic requirements of group key distribution protocols. In addition we defined a set of rank function that were used in our framework and provided the proof of soundness of these functions. The first method in our framework is based on rank theorems to enable and mechanize the verification procedure of security properties for group oriented protocols in PVS theorem proving. Secondly, we provided an inference system defined over rank functions. The approach is based on an elegant and natural proof strategy for the verification of group key protocols. Finally, we introduced an event-B based method based on a formal link between the semantics of group protocols model and event-B. We defined a well-formed connection between event-B invariant and the security property based on conditions to guarantee that the invariant verified in event-B is equivalent to the security property. This latter method is based on an abstracted first-order logic model in order to enable automatic invariant checking for security properties.

Even though we used three different approaches in our framework, we believe that there are some limitations for our framework. In general our methods are based on perfect cryptography conditions. Also, in our methods we tried to target the non-algebraic nature of protocols and analyzed their distributive nature. In addition, the application of the inference system was illustrated on a group protocol in the existence of an active adversary, it will be interesting to investigate the application of the method on similar protocols in the existence of a passive adversary. Also, even the event-B approach is more automated, only invariant properties can be modeled and verified. This is due to the target model and verification tool, namely, event-B and Click’n’Prove.

As future work, we intend to extend some features of our framework in order to be able to apply it on more challenging properties such as key independence (or collusion). Another direction is to provide an implementation of the inference system itself, rather than defining it in a theorem prover. This will provide more flexibility for modeling different protocols, however, If we implement our inference system, then we need to implement some strategies in order to guarantee the success of the verification process. If necessary the user can help the system in order to find an attack. Another open issue is to provide an implementation for the inference system that allows partial proofs based on divergence. For the event-B based approach, it will be more interesting to consider more dynamic properties: forward and backward secrecy, and the most interesting case is key independence. However, in order to achieve this, major modifications of the approach are required to support the distributive nature of the protocol because of the limitations of event-B tools.

References


Abstract. Traditionally, computer simulation techniques are used to perform probabilistic analysis. However, they provide less accurate results and cannot handle large-scale problems due to their enormous CPU time requirements. Recently, a significant amount of formalization has been done in the HOL theorem prover that allows us to conduct precise probabilistic analysis using theorem proving and thus overcome the limitations of the simulation based probabilistic analysis approach. Some major contributions include the formalization of both discrete and continuous random variables and the verification of some of their corresponding probabilistic and statistical properties. This paper presents a concise description of the infrastructures behind these capabilities and the utilization of these features to conduct the probabilistic analysis of real-world systems. For illustration purposes, the paper describes the theorem proving based probabilistic analysis of three examples, i.e., the roundoff error of a digital processor, the Coupon Collector's problem and the Stop-and-Wait protocol.

1 Introduction

Probabilistic analysis is a tool of fundamental importance for the analysis of hardware and software systems. These systems usually exhibit some random or unpredictable elements. Examples include, failures due to environmental conditions or aging phenomena in hardware components and the execution of certain actions based on a probabilistic choice in randomized algorithms. Moreover, these systems act upon and within complex environments that themselves have certain elements of unpredictability, such as noise effects in hardware components and the unpredictable traffic pattern in the case of telecommunication protocols. Due to these random components, establishing the correctness of a system under all circumstances usually becomes impractically expensive. The engineering approach to analyze a system with these kind of unavoidable elements of randomness and uncertainty is to use probabilistic analysis. The main idea behind probabilistic analysis is to mathematically model the random and unpredictable elements of the given system and its environment by appropriate random variables. The probabilistic properties of these random variables are then used to judge system's behavior regarding parameters of interest, such as, downtime, availability, number of failures, capacity, and cost. Thus, instead of guaranteeing that the system meets some given specification under all circumstances, the probability that the system meets this specification is reported.

Even for hardware and software systems for which correctness may be unconditionally guaranteed, the study of system performance primarily relies on probabilistic analysis. In fact, the term system performance commonly refers to the average time required by a system to perform a given task, such as the average runtime of a computational algorithm or the average message delay of a telecommunication protocol. These averages can be computed, based on the probabilistic analysis approach, by using appropriate random variables to model inputs for the system model.

Today, simulation is the most commonly used computer based probabilistic analysis technique. Most simulation softwares provide a programming environment for defining functions that approximate random variables for probability distributions. The random elements in a given system are modeled by these functions and the system is analyzed using computer simulation techniques [6], such as the Monte Carlo Method [20], where the main idea is to approximately answer a query on a probability distribution by analyzing a large number of samples. Statistical quantities, such as average and variance, may then be calculated, based on the data collected during the sampling process, using their mathematical relations in a computer. Due to the inherent nature of simulation, the probabilistic analysis results attained by this technique can never be
termed as 100% accurate. The precision and accuracy of the hardware and software system analysis results has become imperative these days because of the extensive usage of these systems in safety and financial critical areas, such as, medicine, transportation and stock exchange markets. Therefore, simulation cannot be relied upon for the analysis of such systems.

Formal methods [9] are capable of conducting precise system analysis and thus allow us to overcome the above mentioned limitations of the simulation approach. Probabilistic model checking [1, 24] is a rapidly emerging formal probabilistic analysis technique. Like traditional model checking [9], probabilistic model checking involves the construction of a precise state-based mathematical model of the given probabilistic system, which is then subjected to exhaustive analysis to verify if it satisfies a set of formally represented probabilistic properties. Numerous probabilistic model checking algorithms and methodologies have been proposed in the open literature, e.g., [5, 22], and based on these algorithms, a number of tools have been developed, e.g., PRISM [17] and VESTA [25]. Besides the accuracy of the results, the most promising feature of probabilistic model checking is the ability to perform the analysis automatically. On the other hand, it is limited to systems that can only be expressed as probabilistic finite state machines. Another major limitation of the probabilistic model checking approach is state space explosion [4]. Similarly, to the best of our knowledge, it has not been possible to precisely reason about statistical quantities, such as expectation and variance, using probabilistic model checking so far.

The second formal method that can be utilized to conduct probabilistic analysis is higher-order-logic theorem proving. Probabilistic analysis can be conducted within a higher-order-logic theorem prover by first modeling the behavior of the system that needs to be analyzed in higher-order logic, while expressing its random or unpredictable elements in terms of formalized random variables. The second step is to use this formal model to express the probabilistic and statistical properties, regarding the system, as higher-order logic theorems. For this purpose, we require higher-order-logic definitions of probabilistic and statistical properties of random variables, such as, Probability Mass Function (PMF), Cumulative Distribution Function (CDF), expectation and variance, etc. Finally, theorems corresponding to the probabilistic and statistical properties of the system model can be mechanically checked for correctness in a theorem prover.

The above mentioned theorem proving based probabilistic analysis approach tends to overcome the limitations of the simulation and model checking based probabilistic analysis approaches. Due to the formal nature of the models and properties and the inherent soundness of the theorem proving approach, probabilistic analysis carried out in this way will be free from any approximation and precision issues. Similarly, the high expressibility of higher-order logic allows us to analyze a wider range of systems without any modeling limitations, such as the state-space explosion problem in the case of probabilistic model checking, and formally verify analytically complex properties like expectation and variance.

The foremost criteria for implementing a theorem proving based probabilistic analysis framework is to be able to formalize and verify random variables in higher-order logic. Hurd’s PhD thesis [16] can be considered a pioneering work in this regard as it presents a methodology for the formalization and verification of probabilistic algorithms in the higher-order-logic (HOL) theorem prover [8]. Random variables are basically probabilistic algorithms and thus can be formalized and verified, based on their probability distribution properties, using the methodology proposed in [16]. In fact, [16] presents the formalization of some discrete random variables along with their verification, based on the corresponding PMF properties. Building upon Hurd’s formalization framework [16], we have been able to successfully verify the sampling algorithms of a few continuous random variables [13] based on their CDF properties as well. For comparison purposes, it is frequently desirable to summarize the characteristic of the distribution of a random variable by a single number, such as its expectation or variance, rather than an entire function. For example, it is more interesting to find out the expected value of the runtime of an algorithm for an NP-hard problem, rather than the PMF or CDF of the runtime. In [15, 12], we tackled the formalization of expectation and variance in HOL for the first time. We extended Hurd’s formalization framework with a formal definition of expectation, which can be utilized to formalize and verify the expectation and variance characteristics associated with discrete random variables that attain values in positive integers only. The current paper provides a brief overview of the formalization of the above mentioned mathematical foundations. It also illustrates the usage of this available formalization for conducting probabilistic analysis in a theorem prover.
The rest of the paper is organized as follows: Section 2 describes a hypothetical theorem proving based probabilistic analysis framework. This description illustrates how the already formalized mathematical concepts of probability theory fit into the global picture of probabilistic analysis while highlighting some of the interesting future research directions in the area of theorem proving based probabilistic analysis. Then in Sections 3 to 5, we briefly describe the already developed HOL infrastructures for the formalization of discrete random variables [16], the formalization of continuous random variables [13] and the verification of statistical properties [15, 12], respectively. In order to illustrate the practical effectiveness of the proposed approach, we then present the probabilistic analysis of three examples using the HOL theorem prover in Section 6. The examples include the roundoff analysis of a digital processor, the probabilistic analysis of the Coupon Collector’s problem, which is a commercially used algorithm inspired by “Collect all n Coupons and win” contests, and the performance analysis of the Stop-and-Wait protocol, which is a commonly used protocol that ensures reliable communication between computers. Finally, Section 7 concludes the paper.

2 Proposed Framework

A hypothetical model of a theorem proving based probabilistic analysis framework is given in Fig. 1, with some of its most fundamental components depicted with shaded boxes. Like all traditional analysis problems, the starting point of probabilistic analysis is also a system description and some intended system properties and the goal is to check if the given system satisfies these given properties. For simplicity, we have divided system properties into two categories, i.e., system properties related to discrete random variables and system properties related to continuous random variables.

![Fig. 1. Theorem Proving based Probabilistic Analysis Framework](image)

The first step in conducting probabilistic analysis in a theorem prover is to construct a model of the given system in higher-order-logic. For this purpose, the foremost requirement is the availability of infrastructures that allow us to formalize all kinds of discrete and continuous random variables as higher-order-logic functions, which in turn can be used to represent the random components of the given system in its higher-order-logic model. The second step in theorem proving based probabilistic analysis is to utilize the formal model of the system to express system properties as higher-order-logic theorems. The prerequisite for this
step is the ability to express probabilistic and statistical properties related to both discrete and continuous random variables in higher-order-logic. All probabilistic properties of discrete and continuous random variables can be expressed in terms of their PMF and CDF functions, respectively. Similarly, most of the commonly used statistical properties can be expressed in terms of the expectation and variance characteristics of the corresponding random variable. Thus, we require the formalization of mathematical definitions of PMF, CDF, expectation and variance for both discrete and continuous random variables in order to be able to express the given system’s probabilistic and statistical properties as higher-order-logic theorems. The third and the final step for conducting probabilistic analysis in a theorem prover is to formally verify the higher-order-logic theorems developed in the previous step using a theorem prover. For this verification, it would be quite handy to have access to a library of some pre-verified theorems corresponding to some commonly used properties regarding probability distribution functions, expectation and variance. Since, we can build upon such a library of theorems and thus speed up the verification process.

Most of the above mentioned formalization prerequisites of a theorem proving based probabilistic analysis framework have already been fulfilled in the HOL theorem prover, as has been outlined in the last section. These infrastructures and methodologies are briefly described in the next three sections of this paper. On the other hand, to the best of our knowledge, the formalization and verification of statistical properties, like expectation and variance, for continuous random variables is an open research issue as of now. This step requires a higher-order-logic formalization of an integration function that can also handle functions with domains other than real numbers. Lebesgue integration provides this feature and thus the higher-order-logic formalization of some portions of the Lebesgue integration theory [23] can be built upon for formalizing the mathematical concepts of expectation and variance for continuous random variables.

3 Formalization of Discrete Random Variables

A random variable is called discrete if its range, i.e., the set of values that it can attain, is finite or at most countably infinite [26]. Examples of discrete random variables include the outcome of rolling a dice and the number of children in a family. Discrete random variables can be completely characterized by their PMF that returns the probability that a random variable \( X \) is exactly equal to some value \( x \), i.e., \( Pr(X = x) \).

Random variables can be formalized in higher-order-logic as deterministic functions with access to an infinite Boolean sequence \( B^\infty \); source of an infinite random bits with data type \((\text{num} \rightarrow \text{bool})\) [16]. These deterministic functions make random choices based on the result of popping the top most bit in the infinite Boolean sequence and may pop as many random bits as they need for their computation. When the functions terminate, they return the result along with the remaining portion of the infinite Boolean sequence to be used by other functions. Thus, a random variable that takes a parameter of type \( \alpha \) and ranges over values of type \( \beta \) can be represented in HOL by the function

\[
F : \alpha \rightarrow B^\infty \rightarrow \beta \times B^\infty
\]

For example, a \( \text{Bernoulli}(\frac{1}{2}) \) random variable that returns 1 or 0 with equal probability \( \frac{1}{2} \) can be modeled as follows

\[
\vdash \text{bit} = \lambda s. (\text{if shd } s \text{ then 1 else 0}, \text{stl } s)
\]

where the variable \( s \), in the above definition, represents the infinite Boolean sequence and the functions \text{shd} and \text{stl} are the sequence equivalents of the list operation ‘head’ and ‘tail’. The function \text{bit} accepts the infinite Boolean sequence and returns a pair with the first element equal to either 0 or 1 and the second element equal to the unused portion of the infinite Boolean sequence, which in this case is the tail of the sequence.

The work in [16] also presents the formalization of some mathematical measure theory in HOL, which can be used to define a probability function \( \mathbb{P} \) from sets of infinite Boolean sequences to \textit{real} numbers between 0 and 1. The domain of \( \mathbb{P} \) is the set \( \mathcal{E} \) of events of the probability. Both \( \mathbb{P} \) and \( \mathcal{E} \) are defined using the Carathéodory’s Extension theorem, which ensures that \( \mathcal{E} \) is a \( \sigma \)-algebra: closed under complements and
countable unions. The formalized \( P \) and \( E \) can be used to prove probabilistic properties for random variables such as
\[
\vdash P \{ s \mid \text{fst (bit s)} = 1 \} = \frac{1}{2}
\]
where the function \( \text{fst} \) selects the first component of a pair and \( \{ x \mid C(x) \} \) represents a set of all elements \( x \) that satisfy the condition \( C \) in HOL.

The above mentioned infrastructure can be utilized to formalize most of the commonly used discrete random variables and verify their corresponding PMF relations. For example, HOL formalization and verification of Bernoulli and Uniform random variables can be found in [16] and of Binomial and Geometric random variables can be found in [11].

4 Formalization of Continuous Random Variables

A random variable is called continuous if it ranges over a continuous set of numbers [26]. A continuous set of numbers, sometimes referred to as an interval, contains all real numbers between two limits. Many experiments lead to random variables with a range that is a continuous interval. Examples include measuring \( T \), the arrival time of a data packet at a web server (\( S_T = \{ t \mid 0 \leq t < \infty \} \)) and measuring \( V \), the voltage across a resistor (\( S_V = \{ v \mid -\infty < v < \infty \} \)), where \( T \) and \( V \) are both continuous random variables.

The sampling algorithms for discrete random variables are either guaranteed to terminate or satisfy probabilistic termination, meaning that the probability that the algorithm terminates is 1. On the other hand, the formalization of continuous random variables involves non-terminating algorithms and hence require a different approach than discrete random variables.

The work in [13] presents a methodology for the formalization of continuous random variables using the HOL theorem prover. The methodology builds upon Hurd’s verification framework [16], described in the last section, and is primarily based on the concept of the nonuniform random number generation [6], which is the process of obtaining random variates of arbitrary distributions using a Standard Uniform random number generator. The main advantage of this approach is that we only need to formalize one continuous random variable from scratch, i.e., the Standard Uniform random variable, which can be used to model other continuous random variables by formalizing the corresponding nonuniform random number generation method.

Based on the above methodology, [13] presents a framework, illustrated in Fig. 2, for the formalization of all continuous random variables for which the inverse of the CDF can be represented in a closed mathematical form.

![Image: Framework for the Formalization of Continuous Random Variables]

**Fig. 2.** Framework for the Formalization of Continuous Random Variables

The first step in this framework is the formal specification of the Standard Uniform random variable and the formal verification of this definition by proving the corresponding CDF property in the HOL theorem.
Standard Uniform random variable is a continuous random variable for which the probability that it will belong to a subinterval of \([0,1]\) is proportional to the length of that subinterval. It is a well known mathematical fact, see [7] for example, that a Standard Uniform random variate can be modeled by an infinite sequence of random bits (informally coin flips) as follows

\[
\sum_{k=0}^{\infty} \left( \frac{1}{2} \right)^{k+1} X_k
\]  

(1)

where \(X_k\) denotes the outcome of the \(k^{th}\) random bit; \(True\) or \(False\) represented as 1 or 0 respectively. The mathematical relation of Equation (1) presents a sampling algorithm for the Standard Uniform random variable which is quite consistent with formalization methodology, described in the last section, i.e., it allows us to model the Standard Uniform random variable by a deterministic function with access to the infinite Boolean sequence. The specification of this sampling algorithm in higher-order logic is not very straightforward though. Due to the infinite sampling, it cannot be modeled by either of the approaches proposed in [16], i.e., a recursive function or the \textit{probabilistic while loop}. A formalization approach for the Standard Uniform random variable has been presented in [14]. The main idea is to split the mathematical expression of (1) into two steps. The first step is to mathematically represent a discrete version of the Standard Uniform random variable.

\[
(\lambda n. \sum_{k=0}^{n-1} \left( \frac{1}{2} \right)^{k+1} X_k)
\]  

(2)

This lambda abstraction function accepts a positive integer \(n\) and generates an \(n\)-bit Standard Uniform random variable using the computation principle of Equation (1). The continuous Standard Uniform random variable is then represented as a special case of Equation (2) when \(n\) tends to infinity

\[
\lim_{n \to \infty} (\lambda n. \sum_{k=0}^{n-1} \left( \frac{1}{2} \right)^{k+1} X_k)
\]  

(3)

The advantage of expressing the sampling algorithm of Equation (1) in these two steps is that now it can be specified in HOL. The mathematical relationship of Equation (2) can be specified in HOL by a recursive function using the methodology for the formalization of discrete random variables, described in the last section, as it consumes a finite number of random bits, i.e., \(n\). Then, the formalization of the mathematical concept of limit of a \textit{real} sequence [10] in HOL can be used to specify the mathematical relation of Equation (3). The work in [14] also presents the correctness verification of this definition of the Standard Uniform random variable by proving its corresponding CDF relation in HOL.

The second step in the framework for the formalization of continuous probability distributions, given in Fig. 2, is the formalization of the CDF and the verification of its classical properties in HOL. CDF is defined as

\[
F_X(x) = Pr(X \leq x)
\]  

(4)

for any number \(x\), where \(Pr\) represents the probability function. A unique characteristic of CDF is that it can be used to describe the probability distribution of both discrete and continuous random variables. CDF and its properties have been an integral part of the classical probability theory since its early development in the 1930s and play a vital role in characterizing probabilistic properties of random variables. The work in [13] presents a higher-order-logic definition of CDF, based on Equation (4), and utilizes this definition to formally verify the CDF properties, given in Table 1, in the HOL theorem prover.

The next step in the framework for the formalization of continuous probability distributions, given in Fig. 2, is the formal verification of the Inverse Transform Method (ITM) [6], which is a well known nonuniform random generation technique for generating nonuniform random variates for continuous probability distributions for which the inverse of the CDF can be represented in a closed mathematical form. According to the ITM, for any continuous CDF \(F\), the random variable \(X\) defined by \(X = F^{-1}(U)\) has CDF \(F\), where
<table>
<thead>
<tr>
<th>No.</th>
<th>Property</th>
<th>Mathematical Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CDF Bounds</td>
<td>0 ≤ FX(x) ≤ 1</td>
</tr>
<tr>
<td>2</td>
<td>Monotonically Increasing</td>
<td>a &lt; b ⇒ FX(a) ≤ FX(b)</td>
</tr>
<tr>
<td>3</td>
<td>Interval Probability</td>
<td>a &lt; b ⇒ Pr(a &lt; X ≤ b) = FX(b) − FX(a)</td>
</tr>
<tr>
<td>4</td>
<td>CDF at Negative Infinity</td>
<td>FX(−∞) = 0</td>
</tr>
<tr>
<td>5</td>
<td>CDF at Positive Infinity</td>
<td>FX(∞) = 1</td>
</tr>
<tr>
<td>6</td>
<td>CDF is Continuous from the Right</td>
<td>( \lim_{x \to a^+} FX(x) = FX(a) )</td>
</tr>
<tr>
<td>7</td>
<td>CDF Limit from the Left</td>
<td>( \lim_{x \to a^-} FX(x) = Pr(X &lt; a) )</td>
</tr>
</tbody>
</table>

Table 1. CDF Properties

\( F^{-1}(U) \) is defined to be the value of \( x \) such that \( F(x) = U \) and \( U \) represents the Standard Uniform random variable. Mathematically, the ITM can be expressed as

\[
Pr(F^{-1}(U) \leq x) = F(x)
\]  

(5)

In order to verify the above proposition in HOL, the work in [13] presents a formalization of the inverse function of a CDF as a higher-order-logic predicate that accepts two functions, \( f \) and \( g \), of type \( \text{real} \rightarrow \text{real} \) and returns \( \text{True} \) if and only if the function \( f \) is the inverse of the CDF \( g \) according to the ITM proposition. Using this predicate, along with the formal definition of the Standard Uniform random variable, the ITM proposition, given in Equation (5), is verified in HOL. The proof is based on the CDF characteristic of the Standard Uniform random variable and some of the CDF properties, given in Table 1.

At this point, the formalized Standard Uniform random variable can be used to formally specify any continuous random variable for which the inverse of the CDF can be expressed in a closed mathematical form as \( X = F^{-1}(U) \). Whereas, the CDF of this formally specified continuous random variable, \( X \), can be verified, based on simple arithmetic reasoning, using the formal proof of the ITM. For illustration purposes, the work in [13] presents the formal specification of four continuous random variables: Exponential, Uniform, Rayleigh and Triangular. The correctness of these random variables is also verified in [13] by proving their corresponding CDF properties in the HOL theorem prover.

5 Verification of Statistical Properties

In probabilistic analysis, statistical characteristics, like expectation, variance and tail distribution bounds, play a major role in decision making as they tend to summarize the probability distribution characteristics of a random variable in a single number. Due to their widespread interest, the computation of statistical characteristics has now become one of the core components of every modern probabilistic analysis framework.

In this section, we introduce the formalization infrastructure, initially proposed in [11] and presented in Fig. 3, that allows us to formally reason about expectation, variance, and tail distribution properties regarding discrete random variables that attain values in positive integers only.

The first step in the infrastructure, given in Fig. 3, is the formalization of an expression for expectation in higher-order logic. Expectation basically provides the average of a random variable, where each of the possible outcomes of this random variable is weighted according to its probability [2]. The expectation for a function of a discrete random variable, which attains values in the positive integers only, is defined as follows [19].

\[
\text{Ex}_fn[f(X)] = \sum_{n=0}^{\infty} f(n)Pr(X = n)
\]  

(6)

where \( X \) is the discrete random variable and \( f \) represents a function of the random variable \( X \). The above definition only holds if the associated summation is convergent, i.e., \( \sum_{n=0}^{\infty} f(n)Pr(X = n) < \infty \). The expression of expectation, given in Equation (6), has been formalized in [11] as a higher-order-logic function.
using the formalization of the probability function $\mathbb{P}$, explained in Section 3 of this paper. The expected value of a discrete random variable that attains values in positive integers can now be defined as a special case of Equation (6)

$$Ex[X] = Ex(fn[(\lambda n.n)(X)])$$

when $f$ is an identity function. In order to verify the correctness of the above definitions of expectation, they are utilized in [11] to formally verify the following classical expectation properties using the HOL theorem prover.

$$Ex[\sum_{i=1}^{n} R_i] = \sum_{i=1}^{n} Ex[R_i]$$

$$Ex[a + bR] = a + bEx[R]$$

These properties not only verify the correctness of the above definitions but also play a vital role in verifying the expectation characteristics of discrete random components of probabilistic systems, as will be seen in Section 6 of this paper.

The second step in the framework for the formal verification of statistical characteristics, given in Fig. 3, is the formalization of variance and the verification of its properties in the HOL theorem prover. Variance of a random variable $X$ describes the difference between $X$ and its expected value and thus is a measure of its dispersion. It is defined for a discrete random variable, $X$, as follows

$$Var[X] = Ex[(X - Ex[X])^2]$$

The above definition of variance has been formalized in higher-order-logic in [11] by utilizing the formal definitions of expectation, given in Equations (6) and (7). This definition is then formally verified to be correct in the HOL theorem prover by proving the following classical variance properties for it.

$$Var[R] = Ex[R^2] - (Ex[R])^2$$

$$Var[\sum_{i=1}^{n} R_i] = \sum_{i=1}^{n} Var[R_i]$$

Based on the expectation and variance characteristics of a random variable, we can find bounds for the tail distribution, i.e., the probability that a random variable assumes values that are far from its expectation. These bounds are usually calculated using the Markov’s or the Chebyshev’s inequalities [2]. The Markov’s
inequality gives an upper bound for the probability that a non-negative random variable $X$ is greater than or equal to some positive constant.

$$Pr(X \geq a) \leq \frac{Ex[X]}{a} \quad (13)$$

Markov’s inequality gives the best tail bound possible, for a non-negative random variable, using the expectation for the random variable only [21]. This bound can be improved upon if more information about the distribution of the random variable is taken into account. Chebyshev’s inequality is based on this principle and it presents a significantly stronger tail bound in terms of variance.

$$Pr(|X - Ex[X]| \geq a) \leq \frac{Var[X]}{a^2} \quad (14)$$

The third and the fourth steps in the framework for the formal verification of statistical characteristics, given in Fig. 3, are the formal verification of Markov’s and Chebyshev’s inequalities in the HOL theorem prover, respectively. The verification is based on the formal definitions of expectation and variance and their formally verified properties and is outlined in [12].

The above mentioned formalization and verification allows us to reason about expectation, variance and tail distribution properties of any formalized discrete random variable that attains values in positive integers. For illustration purposes, [12] presents the formal verification of the expectation and variance relations for four discrete random variables: Bernoulli, Uniform Binomial and Geometric.

6 Applications

In this section, we illustrate the usage of the formalization, mentioned so far in this paper, for conducting probabilistic analysis. For this purpose, we present the formal probabilistic analysis of three examples using the HOL theorem prover.

6.1 Probabilistic Analysis of Roundoff Error in a Digital Processor

Consider the roundoff error for a particular digital processor to be uniformly distributed over the interval $[-5 \times 10^{-12}, 5 \times 10^{-12}]$. Our goal is to formally verify that the probability of the event when the roundoff error in this digital processor is greater than $2 \times 10^{-12}$ is less than 0.33 and the probability that the final result fluctuates by $\pm 1 \times 10^{-12}$ with respect to the actual value is precisely equal to 0.2.

The first step for formally analyzing the above mentioned properties is to model the randomness with an appropriate random variable in higher-order-logic. The continuous Uniform random variable, formalized using the infrastructure explained in Section 4, can be used for this purpose. The generalized function for the Uniform random variable can be specialized for the interval $[-5 \times 10^{-12}, 5 \times 10^{-12}]$ and the given properties can be expressed and verified as higher-order-logic theorems as follows

$$\vdash Pr\{s \mid 2 \times 10^{-12} < uniform_{rv} \ 5 \times 10^{-12} \ 5 \times 10^{-12} \ s \} < 0.33$$

$$\vdash Pr\{s \mid (-1 \times 10^{-12} < uniform_{rv} \ 5 \times 10^{-12} \ s) \ \wedge \ (uniform_{rv} \ s \ \leq \ 1 \times 10^{-12})\} = 0.2$$

where $uniform_{rv}$ represents the higher-order-logic function corresponding to the Uniform random variable. The proofs are based on CDF properties, given in Table 1, and some basic probability laws, verified in [16].

The above example illustrates the usefulness of formalized continuous random variables in verifying probabilistic quantities with 100% precision.
6.2 Probabilistic Analysis of the Coupon Collector’s Problem

The Coupon Collector’s problem [21] refers to the problem of probabilistically evaluating the number of trials required to acquire all unique, say \( n \), coupons from a collection of multiple copies of these coupons that are independently and uniformly distributed. The problem is similar to the example when each box of cereal contains one of \( n \) different coupons and once you obtain one of every type of coupon, you win a prize.

Our first goal is to verify, using HOL, that the expected value of acquiring all \( n \) coupons is \( nH(n) \), where \( H(n) \) is the harmonic number \( (\sum_{i=1}^{n} 1/i) \). Based on this expectation value, we then reason about the tail distribution properties of the Coupon Collector’s problem using the formally verified Markov’s and Chebyshev’s inequalities.

The Coupon Collector’s problem can be formalized by modeling the total number of trials required to obtain all \( n \) unique coupons, say \( T \), as a sum of the number of trials required to obtain each distinct coupon, i.e., \( T = \sum_{i=1}^{n} T_i \), where \( T_i \) represents the number of trials to obtain the \( i^{th} \) coupon, while \( i-1 \) distinct coupons have already been acquired. The advantage of breaking the random variable \( T \) into the sum of \( n \) random variables \( T_1, T_2, \ldots, T_n \) is that each \( T_i \) can be modeled by the Geometric random variable function.

It is important to note here that the probability of success for these Geometric random variables would be different from one another and would be equal to the probability of finding a new coupon while conducting uniform selection trials on the available \( n \) coupons. Thus, the success probability depends on the number of already acquired coupons and can be modeled using the higher-order-logic function for the discrete Uniform random variable. Based on this methodology, [15] models the Coupon Collector’s problem as a higher-order-logic function, \texttt{coupon}\_\texttt{collector}, that accepts a positive integer greater than 0, \( n+1 \), which represents the total number of distinct coupons that are required to be collected. The function returns the number of trials for acquiring these \( n+1 \) distinct coupons and utilizes the formalized Geometric and Uniform random variables.

Now, using the formal definitions of expectation and variance and the formally verified corresponding properties, given in Section 5, the following properties can be proved in the HOL theorem prover for the above mentioned Coupon Collector function.

\[
\vdash \forall n. \text{expec} \ (\text{coupon}\_\text{collector} \ (n + 1)) = (n + 1) \ (\sum_{i=0}^{n+1} \frac{1}{i+1})
\]

\[
\vdash \forall n \ a. \ 0 < a \Rightarrow P \{ s \mid (\text{fst}(\text{coupon}\_\text{collector} \ (n + 1) \ s)) \geq a \} \leq \frac{(n+1)}{a} \ (\sum_{i=0}^{n+1} \frac{1}{(i+1)})
\]

\[
\vdash \forall n \ a. \ 0 < a \Rightarrow P \{ s \mid \text{abs}((\text{fst}(\text{coupon}\_\text{collector} \ (n + 1) \ s)) - \text{expec} \ (\text{coupon}\_\text{collector} \ (n + 1))) \geq a \} \leq \frac{(n+1)^2}{a^2} \ (\sum_{i=0}^{n+1} \frac{1}{(i+1)})
\]

where \text{expec} and \text{abs} represent the HOL functions for expectation and absolute functions, respectively.

The first theorem gives the expectation of the Coupon Collector’s problem, while the next two correspond to the tail distribution bounds of the Coupon Collector’s problem using Markov and Chebyshev’s inequalities, respectively. The above results exactly match the results of the analysis based on paper-and-pencil proof techniques [21] and are thus 100% precise, which is a novelty that cannot be achieved, to the best of our knowledge, by any existing computer based probabilistic analysis tool.

6.3 Performance Analysis of the Stop-and-Wait Protocol

The Stop-and-Wait protocol [18] utilizes the principles of error detection and retransmission and is a fundamental mechanism for reliable communication between computers. The main idea is that the transmitter keeps on transmitting a data packet (repeating after every \( t_{out} \) units of time) unless and until it receives a valid acknowledgement (ACK) of its reception from the receiver. This section describes the formal verification of the Stop-and-Wait protocol’s average message delay relation for the sake of performance analysis.
Stop-and-Wait protocol is a classical example of a real-time system and thus involves a subtle interaction of a number of distributed processes. The behavior of these processes over time may be specified by higher-order-logic predicates on positive integers [3] that represent the ticks of a clock counting physical time in any appropriate units, e.g., nanoseconds. The granularity of the clock’s tick is believed to be chosen in such a way that it is sufficiently fine to detect properties of interest. Using this methodology, [11] presents a higher-order-logic formalization of the Stop-and-Wait protocol as a logical conjunction of six processes and some initial conditions, which are used to ensure the correct operation of the formal model. The random component in the Stop-and-Wait protocol is channel noise, which is expressed using the formal Bernoulli random variable function.

Now, the formal model of the Stop-and-Wait protocol is used to formally verify the following average message delay relation of the Stop-and-Wait protocol.

\[
\frac{(t_f + t_{out})p}{1 - p} + t_f + t_{prop} + t_{proc} + t_a + t_{prop} + t_{proc} \tag{15}
\]

The variable \(p\), in the above expression, represents the probability of channel error. Whereas, the variables \(t_f, t_a, t_{prop}, t_{proc}\) and \(t_{out}\) denote the time delays associated with data transmission, ACK transmission, message propagation, message processing and time-out delays, respectively. The verification is based on the formalization of expectation and the formally verified expectation properties. Further details about this verification can be found in [11].

It is important to note here that the result of Equation (15) is not new. In fact its existence dates back to the early days of introduction of the Stop-and-Wait protocol. However, it has always been verified using theoretical paper-and-pencil proof techniques, so far. Whereas, the analysis described in this paper is based on mechanical verification using the HOL theorem prover, which is a superior approach to both paper-and-pencil proofs and simulation based analysis techniques.

7 Conclusions

This paper provides a brief overview of the existing work in the HOL theorem prover related to probabilistic analysis. It also highlights the role of each one of these existing methodologies in the area of probabilistic analysis and while doing so presents a hypothetical model of a theorem proving based probabilistic analysis framework. The main idea behind this framework is to use random variables formalized in higher-order logic to model systems, which need to be analyzed, and to verify the corresponding probabilistic and statistical properties in a theorem prover. Because of the formal nature of the models, the analysis is free of approximation and precision errors and due to the high expressive nature of higher-order logic a wider range of systems can be analyzed. Thus, the theorem proving based probabilistic analysis approach can prove to be very useful for the performance and reliability optimization of safety critical and highly sensitive engineering and scientific applications.

We utilized the above mentioned mathematical foundations to present the formal probabilistic analysis of three examples, i.e., the Roundoff error in a digital processor, the Coupon Collectors Algorithm and the Stop-and-Wait protocol. The analysis results exactly matched the results obtained by paper-and-pencil proof techniques and are thus 100% precise. The successful handling of these diverse probabilistic analysis problems by the proposed approach clearly demonstrates its feasibility for real-world probabilistic analysis issues.

The main limitation of the proposed approach is the associated significant user interaction, i.e., the user needs to guide the proof tools manually since we are dealing with higher-order logic, which is known to be non-decidable. On the other hand, simulation is capable of handling all sorts of probabilistic analysis problems in an automated way but the solutions provided are not exact. Whereas, probabilistic model is capable of providing exact answers for a subset of probabilistic analysis problems. We believe that all these three techniques have to play together in order to form a successful probabilistic analysis framework. For example, an efficient approach would be to use simulation for the less critical parts of the analysis, model checking for the critical parts that it can handle and theorem proving for the remaining critical parts.
Finally, it is important to note that the presented methodologies and frameworks are not specific to the HOL theorem prover and can be adapted to any other higher-order-logic theorem prover, such as Isabelle, Coq or PVS, as well.

References

Checking the Satisfiability of XML-Specifications

Dr. Harald Hiss
University Freiburg, D-79110 Freiburg, Germany, hims@informatik.uni-freiburg.de,
WWW home page: http://dbis.informatik.uni-freiburg.de/index.php?person=hiss

Abstract. New developments in databases build on XML-technologies. Concepts of the relations model are transferred to XML. This is not unproblematic, transfer of integrity constraints causes a problem. Some XML-specifications are unsatisfiable.

A deductive checker is presented. An extensive formalization developed with Isabelle integrates circular XML-specifications with an inductive method. These XML-specifications are unsatisfiable. The checker generates a representation with F-Logic that is checked with Florid. The correctness is proven.

1 Checking the Satisfiability

New developments in databases build on XML [BMP+06] technologies. Concepts of the relational [AHV95] model are transferred to XML. This is not unproblematic, transfer of integrity constraints causes a problem. Some XML-specifications are unsatisfiable.

A deductive checker for XML-specifications is presented. The complexity of the satisfiability is proven in [FL02]. Implication of relational integrity that is undecidable [CV85] is represented with XML-specifications. A transformation for model checking XML-specifications is presented in [His07]. The transformation generates constraints. A model checker proves the satisfiability of the constraints. An extensive formalization developed with Isabelle [Pau94b] proves the correctness. Circular XML-specifications are integrated with an inductive method [Pau94a]. These XML-specifications are unsatisfiable. A deductive checker is presented based on this development. The checker generates a representation with F-Logic [Kre95] that is checked with Florid [HLS07]. The correctness of the checker is proven.

XML-specifications are introduced in the next section with a database of teachers. Section 3 presents a formalization of XML-specifications illustrated with the example. Then section 4 formalizes circular XML-specifications. Section 5 presents theorems for proving that circular XML-specifications are unsatisfiable in section 6. Then section 7 presents the deductive checker and section 8 concludes the contribution.

2 A Database of Teachers

A database of teachers is represented with an XML-specification. Elements (teachers, research, subject) and attributes (name, instructor) are defined with the structural schema in figure 1. Content models form a structure for XML-trees. The root labeled teachers stores content model teacher+. XML-trees of the structural schema have a teachers root with teacher children. Figure 2 shows an instance, the next section presents details. Attribute instructor represents a teacher. Integrity represents dependencies of attributes. Keys and inclusion constraints formalize integrity. Key teacher.name → teacher represents teacher with name. Inclusion constraint subject.instructor ⊆ teacher.name represents the dependency of instructor of subject on names.

\[ \text{teacher.name} \rightarrow \text{teacher} \]
\[ \text{subject.instructor} \rightarrow \text{subject} \]
\[ \text{subject.instructor} \subseteq \text{teacher.name} \]

The XML-tree presented in figure 2 satisfies teacher.name → teacher. The teacher nodes store Dr. Brett and Prof. Crey. The instructors are contained, subject.instructor ⊆ teacher.name is satisfied. several subjects store Dr. Brett, the key subject.instructor → subject isn’t satisfied. The example is unsatisfiable. The
Fig. 1. A graph represents the structural schema of the example.

Fig. 2. An example XML-tree stores the teachers Dr. Brett and Prof. Crey.
structural schema contains a branch. The teacher nodes, formalized with $ext(teacher)$, have more subject descendants.
\[2|ext(teacher)| \leq |ext(subject)|\]
A document has at least a teacher.
\[|ext(teacher)| < |ext(subject)|\]
A contradiction is proven with $subject.instructor \rightarrow subject$ and $subject.instructor \subseteq teacher.name$. The next section presents the formalization of XML-specifications that forms the fundament for the checker presented in section 7.

### 3 Formalization of the Database

The section presents a formalization of XML-specifications illustrated with the database of teachers. The structural schema and integrity are formalized. Attributes $A$ ($name$, $instructor$) and elements $E$ ($teachers$, $subject$) with root $r$ are defined with a structural schema [BMP06]. The example in figure 3 has the root

```xml
<!DOCTYPE teachers [
<!ELEMENT teachers (teacher*)>
<!ELEMENT teacher (teach, research)>
<!ELEMENT teach (subject, subject)>
<!ELEMENT research (#PCDATA)>
<!ELEMENT subject (#PCDATA)>
<!ATTLIST teacher name CDATA #REQUIRED>
<!ATTLIST subject instructor CDATA #REQUIRED>
<!ATTLIST subject instructor CDATA #REQUIRED> ]>
```

**Fig. 3.** The structural schema of figure 1 is defined.

teachers. The attributes of an element are stored with function $R$, teacher stores attribute name. Function $P$ stores the content models. Element teachers stores teacher+. Regular expressions [HMU06] are formalized with an inductive method [BW]. Concatenation, choice, star, plus and question mark form content models with labels $r \in E$, text $S$ and empty content $\epsilon$.

$$\alpha ::= \epsilon \mid S \mid \tau \mid (\alpha, \alpha) \mid (\alpha|\alpha) \mid \alpha^* \mid \alpha^+ \mid \alpha?$$

Wellformed structural schemas are formalized. The sets $A$ and $E$ are disjoint and don’t include $S$. The functions $P$ and $R$ are defined for $E$. Labels in $(E \cup \{S\}) \setminus \{r\}$ form content models that connect $r$ with the elements. XML-trees are formalized with the nodes $V$. The example XML-tree in figure 4 includes the nodes $v_1, v_2, ..., v_9$. Function lab stores labels in $A \cup E \cup \{S\}$, teacher is stored for $v_2$ and name for $v_4$. Children are stored with $ele$. Parents are unique. Nodes $[v_2, v_3]$ are the children of $v_1$ that represents root, the only node with the label $r$. Attribute nodes are stored with $att$. The example defines name for teacher. For $v_2$ and this attribute $att$ stores $v_4$. Function val stores text of nodes with a label in $A \cup \{S\}$, $v_4$ stores Dr. Brett. Text nodes don’t have children. Label name proves $v_4$ doesn’t have children.

$$ext(\tau) = \{ v \mid v \in V \land lab v = \tau \land \tau \in E \cup \{S\} \}$$

Nodes labeled $\tau \in E \cup \{S\}$ are formalized with $ext(\tau)$. For example, $ext(teacher)$ stores $\{v_2, v_3\}$. Paths are formalized with an inductive method [Pau94a].

$$v_1 \in ext(\tau) \quad \text{path}(v_1, v_1) \quad \text{path}(v_1, v_3) \quad v_2 \in ele \ v_3 \quad \text{path}(v_1, v_2)$$
Paths are reflexive and $\text{path}(v_1, v_3)$ can be extended with children of $v_3$. The example satisfies $\text{path}(v_1, v_1)$, $\text{path}(v_1, v_2)$ and $\text{path}(v_1, v_3)$. XML-trees don’t have cycles. Paths connect root with the element nodes. The section formalizes the validation of XML-trees. Children are labeled with the content models.

$$\text{parse } B \left( \text{grammar}(\text{lab } v) \right) \land \text{getWord}(B) = (\text{map } \text{lab } (\text{ele } v))$$

An element node $v$ has a parse tree $B$ for the formal grammar [HMU06], formalized with $\text{grammar}(\text{lab } v)$. The labels of $B$ computed with $\text{getWord}(B)$ are equal to the labels of the children. The nodes $v_5$, $v_6$, the children of $v_2$, have the following labels.

$$\text{map } \text{lab } (\text{ele } v_2) = [\text{teach}, \text{research}]$$

The grammar for teacher accepts the labels. Details of the formalization are presented in [His07]. The section formalizes integrity. Attribute $l$ of a $τ$ node $v$ is stored with $v.l = \text{val}(\text{att}(v, l))$. The example stores Prof. Crey with $v_9$.name. Attributes $L = < l_1, ..., l_n >$ are stored with $v[L] = < v.l_1, ..., v.l_n >$. The $L$ tuples of $τ$ nodes are formalized with $\text{ext}(τ[L])$ and $\text{ext}(τ.l)$ for one attribute. With $\text{ext}$(teacher.name), the example stores {Dr. Brett, Prof. Crey}.

$$\text{ext}(τ[L]) = \{ v[L] | v \in V \land \text{lab } v = τ \}$$

The section formalizes integrity. Key $τ[L] → τ$ identifies $τ$ nodes with attributes $L$. The formalization considers a function $f(v) = v[L]$.

$$τ[L] → τ \iff \text{inj}_\text{on } f \text{ ext}(τ)$$

For $τ \in E$ with attributes $L$, the key is satisfied provided $f$ is injective for the $τ$ nodes. The example satisfies teacher.name → teacher. Inclusion constraints represent dependencies of attributes.

$$τ_1[L_1] \subseteq τ_2[L_2] \iff \text{ext}(τ_1[L_1]) \subseteq \text{ext}(τ_2[L_2])$$

An XML-tree satisfies inclusion constraint $τ_1[L_1] \subseteq τ_2[L_2]$ whenever $L_1$ tuples of $τ_1$ nodes depend on $L_2$ tuples of $τ_2$. The formalization has been implemented with Isabelle [NP07]. The formalization is the fundament for the contribution. The next section formalizes circular XML-specifications.

4 Circular XML-Specifications

The section presents the formalization of circular XML-specifications. They are formalized with an inductive method [Pau94a] that proves the correctness of cryptographic protocols in [Pau98]. The section formalizes
ways. A branch has two ways to a descendant. XML-specifications are circular when there is a branch without cycle and the descendant of the branch depends on the ancestor. The next section proves that circular XML-specifications are unsatisfiable. The branch proves a constraint and the dependency proves the opposite. Section 7 presents a deductive checker for XML-specifications based on circular XML-specifications. The formalization considers normalized content models [His07].

$$\forall \tau \in E. (P \tau = \epsilon) \lor (\exists \tau_1, \tau_2 \in E \cup \{S\}, P \tau = \tau_1 \lor P \tau = (\tau_1, \tau_2) \lor (P \tau = (\tau_1, \tau_2) \land \tau_1 \neq \tau_2))$$

They have less or equal two labels and don’t contain plus, question mark and star. The section formalizes ways formed with content models.

$$\begin{array}{c}
\tau_1 \in E \\
way(\tau_1, \tau_1)
\end{array} \quad \begin{array}{c}
P \tau_1 = \tau_3 \\
way(\tau_1, \tau_2)
\end{array}$$

Ways are reflexive and content $\tau_3$ of $\tau_1$ with way($\tau_3, \tau_2$) implies way($\tau_1, \tau_2$).

$$\begin{array}{c}
P \tau_1 = (\tau_3, \tau_4) \\
way(\tau_1, \tau_2)
\end{array} \quad \begin{array}{c}
P \tau_1 = (\tau_4, \tau_3) \\
way(\tau_1, \tau_2)
\end{array}$$

Ways can be extended with concatenated content when $\tau_3$ proves way($\tau_3, \tau_2$).

$$\begin{array}{c}
P \tau_1 = (\tau_3|\tau_4) \\
way(\tau_1, \tau_2)
\end{array} \quad \begin{array}{c}
P \tau_1 = (\tau_3, \tau_2) \\
way(\tau_1, \tau_2)
\end{array} \quad \begin{array}{c}
P \tau_1 = (\tau_3, \tau_2) \\
way(\tau_1, \tau_2)
\end{array}$$

Content models ($\tau_3|\tau_4$) extend ways where the elements $\tau_3, \tau_4$ have a way. The $\tau_1$ nodes have $\tau_2$ descendants when there is a way from $\tau_1$ to $\tau_2$. Section 6 proves an extensive library of theorems.

$$\text{branch}(\tau_1, \tau_2) \iff \exists \tau_3, \tau_4, \tau_5 \in E. \text{way}(\tau_1, \tau_3) \land P \tau_3 = (\tau_4, \tau_5) \land \text{way}(\tau_4, \tau_2) \land \text{way}(\tau_5, \tau_2)$$

A structural schema has a branch from $\tau_1$ to $\tau_2$ when there is a way from $\tau_1$ to an element $\tau_3$ such that the labels of the concatenated content model have a way to the descendant. The element $\tau_3$ represents the branch. An XML-tree of a structural schema with the contents branch($\tau_1, \tau_2$) includes $\tau_2$ descendants for $\tau_1$ nodes. The example has a branch, teacher and teach have a way to subject. Element teach represents the branch with (subject, subject). The example satisfies branch(teacher, subject) and branch(subject, subject).

The section formalizes cycles. Circular XML-specifications have a branch without cycle. Elements that have a possible way are formalized.

$$\begin{array}{c}
\tau_2 \in P \tau_1 \\
\text{possibleWay}(\tau_1, \tau_2)
\end{array} \quad \begin{array}{c}
\text{possibleWay}(\tau_1, \tau_3) \\
\tau_2 \in P \tau_3
\end{array}$$

Possible ways are proven with content models. A possible way is obtained with an element $\tau_3 \in P \tau_1$ and a possible way from $\tau_3$ to $\tau_2$. XML-trees of a structural schema with possibleWay($\tau_1, \tau_2$) possibly have a $\tau_2$ descendant for a $\tau_1$ node. The example has a possible way from teacher to subject. Possible ways formalize cycles.

$$\text{cycle}(\tau) \iff \text{possibleWay}(\tau, \tau)$$

Structural schemas that satisfy possibleWay($\tau, \tau$) have a cycle with $\tau$. The example doesn’t have a cycle. The section formalizes integrity that bounds elements.

$$\text{anchor}(\tau_1, \tau_2) \iff \exists L_1 \subseteq (R \tau_1). \exists L_2 \subseteq (R \tau_2). \tau_1[L_1] \rightarrow \tau_1 \land \tau_1[L_1] \subseteq \tau_2[L_2]$$

A key $\tau_1[L_1] \rightarrow \tau_1$ and an inclusion constraint form an anchor when $\tau_1[L_1]$ depends on $\tau_2[L_2]$. The example has an anchor from subject to teacher.

$$\text{onceOccurs}(\tau_1, \tau_2) \iff P \tau_2 = \tau_1 \lor \exists \tau_3. \tau_3 \neq \tau_1 \land (P \tau_2 = (\tau_1, \tau_3) \lor P \tau_2 = (\tau_3, \tau_1))$$

37
Single and concatenated content models that contain a particular label once are formalized with \( \text{onceOccurs} \). The example satisfies \( \text{onceOccurs}(\text{teacher}, \text{teach}) \).

\[
\text{moreOccurs}(\tau_1) \iff \exists \tau_2, \tau_4 \in E. \tau_1 \in P \tau_2 \land \tau_1 \in P \tau_4 \land \tau_2 \neq \tau_4
\]

A structural schema satisfies \( \text{moreOccurs}(\tau_1) \) when \( \tau_1 \) occurs in the content models of some elements. The example satisfies \( \text{moreOccurs}(S) \) because \( \text{research} \) and \( \text{subject} \) store text. The section presents the formalization of bounds.

\[
\begin{align*}
\text{anchor}(\tau_1, \tau_2) & \quad \text{onceOccurs}(\tau_1, \tau_2) \quad \neg \text{moreOccurs}(\tau_1) \\
\text{bounds}(\tau_1, \tau_2) & \quad \text{bounds}(\tau_1, \tau_2)
\end{align*}
\]

Element \( \tau_2 \) bounds \( \tau_1 \) when integrity forms an anchor. The example bounds \( \text{subject} \) with \( \text{teacher} \). Element \( \tau_2 \) that stores \( \tau_1 \) once bounds \( \tau_1 \).

\[
\begin{align*}
\text{way}(\tau_1, \tau_2) & \quad \neg \text{cycle}(\tau_1) \\
\text{bounds}(\tau_1, \tau_2) & \quad \text{bounds}(\tau_1, \tau_3) \quad \text{bounds}(\tau_3, \tau_2) \\
\text{bounds}(\tau_1, \tau_2) & \quad \text{bounds}(\tau_1, \tau_2)
\end{align*}
\]

A way without cycle satisfies \( \text{bounds}(\tau_1, \tau_2) \). The relation is transitive.

\[
\text{circular} \iff \text{way}(r, \tau_1) \land \text{branch}(\tau_1, \tau_2) \land \neg \text{cycle}(\tau_1) \land \text{bounds}(\tau_2, \tau_1)
\]

An XML-specification is circular, when the structural schema has a way from \( r \) to a branch that doesn’t have a cycle at the ancestor of the branch and the descendant bounds the ancestor. The example is circular. There is a way from \( r \) to \( \text{teacher} \) and a branch from \( \text{teacher} \) to \( \text{subject} \). The ancestor \( \text{teacher} \) doesn’t have a cycle. The descendant \( \text{subject} \) is bounded with \( \text{teacher} \). The section has presented the formalization of circular XML-specifications. The next section proves theorems for proving the correctness of the checker presented in section 7.

## 5 Paths in XML-Trees

The previous section has formalized circular XML-specifications based on the formalization in section 3. This section formalizes descendants and proves theorems of paths and ways. The next section proves that circular XML-specifications are unsatisfiable. The theorems prove the correctness of the checker presented in section 7.

Descendants are formalized with an inductive method [Pau94a] that formalizes circular XML-specifications in section 4. Then the section presents theorems for proving that the descendants of a branch of an XML-tree are disjoint in the next section. The \( \tau_2 \) descendants of \( \tau_1 \) nodes are formalized with \( \text{descendant}(\tau_1, \tau_2) \).

\[
\begin{align*}
v_1 \in \text{ext}(\tau_1) & \quad v_1 \in \text{ext}(\tau_2) \\
v_1 \in \text{descendant}(\tau_1, \tau_1) & \quad v_1 \in \text{descendant}(\tau_1, \tau_2) \\
v_1 \in \text{descendant}(\tau_1, \tau_3) & \quad v_1 \in \text{descendant}(\tau_1, \tau_2)
\end{align*}
\]

The descendants are reflexive for \( \tau_1 \) nodes. Node \( v \) is a descendant of a \( \tau_1 \) node when the parent \( v_3 \in \text{ext}(\tau_3) \) is a descendant. The \( \text{subject} \) nodes of the example in figure 2 are descendants of teachers. They are children of a \( \text{teach} \) node that is a child of a \( \text{teacher} \) node of \( \text{descendant}(\text{teacher}, \text{teacher}) \). Next, paths without label \( \tau \) are formalized with \( \text{differentLabel} \).

\[
\begin{align*}
\text{differentLabel}(v_1, v_1, \tau) & \quad \text{lab} v_2 \neq \tau \quad v_2 \in \text{ele} v_3 \quad \text{differentLabel}(v_1, v_3, \tau) \\
\text{differentLabel}(v_1, v_2, \tau) & \quad \text{differentLabel}(v_1, v_1, \tau)
\end{align*}
\]

Nodes \( v_1 \) with a label \( \tau_1 \) not equal \( \tau \) satisfy \( \text{differentLabel}(v_1, v_1, \tau) \). Such paths are extended with nodes having a label unequal \( \tau \). The section formalizes next nodes of a specified label.

\[
\text{same}(v_1, v_2, \tau) \iff v_1 = v_2 \land v_1 \in \text{ext}(\tau)
\]
The \( \tau \) nodes satisfy same.

\[
\text{nextDifferent}(v_1, v_2, \tau) \iff \exists v_3. \ v_2 \in \text{ext}(\tau) \land v_2 \in \text{ele} \ v_3 \land \text{differentLabel}(v_1, v_3, \tau)
\]

The section formalizes paths to \( \tau \) nodes that don’t contain \( \tau \).

\[
\text{next}(v_1, v_2, \tau) \iff \text{same}(v_1, v_2, \tau) \lor \text{nextDifferent}(v_1, v_2, \tau)
\]

Nodes that are next have a path. The section formalizes descendants of the branch.

\[
v \in \text{descendant1}(\tau_1, \tau_2, \tau_3) \iff \exists v_1, v_2, v_3, v_4. \ v_1 \in \text{ext}(\tau_1) \land \text{next}(v_1, v_2, \tau_2) \land \text{ele} \ v_2 = [v_3, v_4] \land \text{next}(v_3, v, \tau_3)
\]

A node \( v \in \text{descendant1}(\tau_1, \tau_2, \tau_3) \) takes the first way at the next \( \tau_2 \) descendant of a \( \tau_1 \) node. Node \( v \) is the next \( \tau_3 \) descendant of the first child of the \( \tau_2 \) descendant. Maths and chemistry are stored with \( \text{descendant1}(\text{teacher}, \text{teach}, \text{subject}) \).

\[
v \in \text{descendant2}(\tau_1, \tau_2, \tau_3) \iff \exists v_1, v_2, v_3, v_4. \ v_1 \in \text{ext}(\tau_1) \land \text{next}(v_1, v_2, \tau_2) \land \text{ele} \ v_2 = [v_3, v_4] \land \text{next}(v_4, v, \tau_3)
\]

Nodes in \( \text{descendant2}(\tau_1, \tau_2, \tau_3) \) take the second way to \( \tau_3 \). The section presents theorems of paths and ways. Then it is proven that circular XML-specifications are unsatisfiable.

\[
\frac{v_2 \in \text{ele} \ v_1 \quad v_3 \in \text{ele} \ v_1 \quad v_2 \neq v_3}{\neg \text{path}(v_2, v_3)} \quad T_1
\]

Theorem \( T_1 \) proves that children aren’t connected. The proof assumes the opposite, an induction with \( \text{path}(v_2, v_3) \) proves the following.

\[
P_1(v_2, v_3) \iff v_2 \in \text{ele} \ v_1 \land v_3 \in \text{ele} \ v_1 \land v_2 \neq v_3 \rightarrow \text{false}
\]

The base case proves a contradiction with \( v_2 \neq v_3 \). The induction step considers a node \( v_4 \) with the child \( v_3 \) and \( \text{path}(v_2, v_4) \), \( P_1(v_2, v_3) \) is proven. Parents are unique, \( v_1 \) is equal to \( v_4 \). XML-trees don’t have cycles. A contradiction is proven with \( v_2 \in \text{ele} \ v_1 \) and \( \text{path}(v_2, v_1) \).

\[
\frac{\text{path}(v_1, v_3) \quad \text{path}(v_2, v_3)}{\text{path}(v_1, v_2) \lor \text{path}(v_2, v_1)} \quad T_2
\]

Nodes with a path to the same node are connected (\( T_2 \)). The contrapositive is proven. An induction with \( \text{path}(v_1, v_3) \) proves that \( v_2 \) doesn’t have a path to \( v_3 \).

\[
P_2(v_1, v_3) \iff \neg \text{path}(v_1, v_2) \land \neg \text{path}(v_2, v_1) \rightarrow \neg \text{path}(v_2, v_3)
\]

The hypothesis with one node is a tautology. The induction step considers a node \( v_4 \) with \( \text{path}(v_1, v_4) \) that satisfies \( P_2(v_1, v_4) \). Node \( v_4 \) has the child \( v_3 \) and \( P_2(v_1, v_3) \) is proven. When \( v_2 \) is equal to \( v_3 \), it is a child of \( v_4 \) and the path from \( v_1 \) to \( v_4 \) gives a contradiction with \( \neg \text{path}(v_1, v_2) \). Otherwise, \( \text{path}(v_2, v_3) \) proves a path from \( v_2 \) to \( v_4 \) with child \( v_3 \). Then \( P_2(v_1, v_4) \) proves a contradiction.

\[
\frac{\text{path}(v_1, v_2) \quad \text{lab} \ v_1 = \tau_1 \quad \text{lab} \ v_2 = \tau_2 \quad v_1 \neq v_2}{\text{possibleWay}(\tau_1, \tau_2)} \quad T_3
\]

Paths prove a possible way (\( T_3 \)). An induction with \( \text{path}(v_1, v_2) \) proves this.

\[
P_3(v_1, v_2) \iff \forall \tau_1, \tau_2. \ \text{lab} \ v_1 = \tau_1 \land \text{lab} \ v_2 = \tau_2 \land v_1 \neq v_2 \rightarrow \text{possibleWay}(\tau_1, \tau_2)
\]
The base case proves a contradiction. The induction step satisfies \( P_3(v_1, v_3) \) and considers a child \( v_2 \) of \( v_3 \). The section proves that \( \tau_1 \) has a possible way to the label of \( v_2 \). When \( v_1 \) equals \( v_3 \) the content model of \( \tau_1 \) includes \( \tau_2 \), a possible way is proven. Otherwise, the induction hypothesis proves possibleWay(\( \tau_1, (\text{lab } v_3) \)). The content model of the label of \( v_3 \) includes \( \tau_2 \).

\[
\begin{align*}
\text{next}(v_1, v_2, \tau) & \quad \text{next}(v_1, v_3, \tau) & \quad \text{path}(v_2, v_3) \\
\quad v_2 = v_3
\end{align*}
\]

Theorem \( T_4 \) proves that nodes are equal when they are connected next descendants of the same node. Nodes that are equal can satisfy same. Otherwise, \( v_2 \) and \( v_3 \) have a path without \( \tau \) from \( v_1 \). Then \( v_1 \) has a differentLabel path to the parent of \( v_3 \). The path contains the \( \tau \) node \( v_2 \).

\[
\begin{align*}
v_1 \in \text{ext}(\tau_1) & \quad \text{way}(\tau_1, \tau_2) \\
\exists v_2 \in \text{ext}(\tau_2), \text{path}(v_1, v_2) \\
\end{align*}
\]

With \( T_5 \), a path to a \( \tau_2 \) node is proven for a \( \tau_1 \) node when there is a way from \( \tau_1 \) to \( \tau_2 \). An induction with \( \text{way}(\tau_1, \tau_2) \) uses hypothesis \( P_4(\tau_1, \tau_2) \).

\[
P_4(\tau_1, \tau_2) \iff \forall v_1 \in \text{ext}(\tau_1). \exists v_2 \in \text{ext}(\tau_2). \text{path}(v_1, v_2)
\]

The base case is proven with one node. The induction step considers the content models of \( \tau_1 \). Concatenated content models prove a \( \tau_3 \) child that extends a path to \( v_2 \) proven with hypothesis \( P_4(\tau_3, \tau_2) \). Otherwise \( P \tau_1 = (\tau_3[\tau_4]) \), a child in \( \text{ext}(\tau_3) \cup \text{ext}(\tau_4) \) is proven. Then \( P_4(\tau_3, \tau_2) \) and \( P_4(\tau_4, \tau_2) \) prove a path. Theorems have been presented for proving that circular XML-specifications are unsatisfiable in the next section. Section 7 presents a deductive checker.

### 6 Unsatisfiable Circular XML-Specifications

The section proves that circular XML-specifications are unsatisfiable with the theorems of the previous section. A branch proves more nodes of the descendant. The branch is bounded, a contradiction is proven.

\[
\begin{align*}
\text{branch}(\tau_1, \tau_2) & \quad \text{ext}(\tau_1) \neq \emptyset & \quad \neg \text{cycle}(\tau_1) \\
|\text{ext}(\tau_1)| < |\text{ext}(\tau_2)|
\end{align*}
\]

An XML-tree with \( \tau_1 \) nodes contains more descendants of a branch when the structural schema doesn’t have a cycle with \( \tau_1 \). The section proves the first and second descendants of the XML-tree are disjoint. Then a cycle with \( \tau_1 \) is proven.

The proof considers \( \tau_3 \) that represents the branch. It is proven that the intersection of descendant1(\( \tau_1, \tau_3, \tau_2 \)) and descendant2(\( \tau_1, \tau_3, \tau_2 \)) is empty. A node \( v \) of the intersection is presumed. Function \( f_1 \) (\( f_2 \)) chooses the ancestor of the first (second) descendant of a branch of the XML-tree. The function \( f_1(v) = v_1 \) is defined with nodes \( v_2 \), \( v_3 \) and \( v_4 \) that satisfy next(\( v_1, v_2, \tau_3 \)), ele \( v_2 = [v_3, v_4] \) and next(\( v_3, v, \tau_2 \)). In this way, \( f_2(v) = v_5 \) is defined with nodes \( v_5 \), \( v_7 \) and \( v_8 \) that satisfy next(\( v_5, v_6, \tau_3 \)), ele \( v_6 = [v_7, v_8] \) and next(\( v_8, v, \tau_2 \)). When \( v_1 \neq v_5 \), \( T_3 \) proves possibleWay(\( \tau_1, \tau_1 \)) with the path of \( v_1 \) and \( v_3 \) proven with \( T_2 \). This is a contradiction with cycle(\( \tau_1 \)). Thus, \( v_1 \) and \( v_5 \) are equal. The proof considers node \( v_2 \) (\( v_6 \)) that represents the first (second) branch. They have a path to \( v \). Theorem \( T_2 \) proves a path connects them. The nodes are next \( \tau_3 \) nodes of \( v_1 \), \( T_4 \) proves they are equal. Moreover, the children are equal. They aren’t connected (\( T_1 \)) and have the descendant \( v \), \( T_2 \) proves a contradiction. The first and second descendants are disjoint.

An XML-tree has more \( \tau_2 \) descendants than descendants of the first branch.

\[
|\text{descendant}(\tau_1, \tau_2)| \geq |\text{descendant1}(\tau_1, \tau_3, \tau_2) \cup \text{descendant2}(\tau_1, \tau_3, \tau_2)|
\]

The \( \tau_2 \) descendants contain the descendants of a branch.

\[
|\text{descendant1}(\tau_1, \tau_3, \tau_2)| + |\text{descendant2}(\tau_1, \tau_3, \tau_2)| > |\text{descendant1}(\tau_1, \tau_3, \tau_2)|
\]
They are disjoint, the sum is considered. The XML-tree has $\tau_1$ nodes, $T_5$ proves descendants of the branch. The proof presumes less or equal $\tau_2$ descendants than $\tau_1$ nodes. The previous inequation proves more $\tau_1$ nodes than $\tau_2$ descendants of the first branch. A function $f_3$ chooses a first descendant of a branch. The function is defined with $f_3(v_1) = v_2$ and nodes $v_3, v_4$ and $v_5$ such that $\text{next}(v_1, v_3, \tau_3)$, $\text{ele} v_3 = [v_4, v_5]$ and $\text{next}(v_4, v_2, \tau_2)$ are satisfied. The range equals the first descendants. $T_5$ proves $f_3$ is defined for $\tau_1$ nodes. The domain is greater, there are nodes $v_1, v_2 \in \text{ext}(\tau_1)$ with the descendant $v_3 = f_3(v_1) = f_3(v_2)$. The nodes have a path to $v_3$, $T_2$ proves $v_1$ and $v_2$ are connected. Then $T_3$ proves possibleWay$(\tau_1, \tau_1)$. This is a contradiction with $\neg \text{cycle}(\tau_1)$. The XML-tree satisfies $|\text{ext}(\tau_1)| < |\text{descendant}(\tau_1, \tau_2)|$. The descendants are contained in $\text{ext}(\tau_2)$, the theorem is proven. Theorem $T_5$ proves $\tau_1$ nodes with $\text{way}(r, \tau_1)$. Circular XML-specifications satisfy $|\text{ext}(\tau_1)| < |\text{ext}(\tau_2)|$. The next theorem proves the opposite with $\text{bounds}(\tau_2, \tau_1)$. Circular XML-specifications are unsatisfiable.

$$\text{bounds}(\tau_1, \tau_2) \leq |\text{ext}(\tau_2)|$$

An induction with $\text{bounds}(\tau_1, \tau_2)$ proves less or equal $\tau_1$ than $\tau_2$ nodes. Anchors prove the inequation with integrity. A constraint of the transformation for model checking XML-specifications presented in [His07] proves an equation for elements that occur once. An injective function chooses a descendant for ways without cycle. Finally, the induction hypothesis proves the theorem. The proof defines the induction hypothesis $|\text{ext}(\tau_1)| \leq |\text{ext}(\tau_2)|$. Integrity implies inequations proven in [His07]. An anchor is defined with a key $\tau_1[L_1] \rightarrow \tau_1$ and an inclusion constraint $\tau_1[L_1] \subseteq \tau_2[L_2]$.

$$|\text{ext}(\tau_1)| = |\text{ext}(\tau_1[L_1])| \leq |\text{ext}(\tau_2[L_2])| \leq |\text{ext}(\tau_2)|$$

The key proves the number of $\tau_1$ nodes and $L_1$ tuples is equal. The inclusion constraint proves that they are less or equal than the $L_2$ tuples of $\tau_2$. Then the proof considers elements that occur once. The transformation proves a structured representation of nodes.

$$|\text{ext}(\tau_1)| = \sum_{\tau_1 \in P} |\text{children}(\tau_2, \tau_1, i)|$$

The constraint represents the $\tau_1$ nodes with the first and second children.

$$|\text{ext}(\tau_1)| = |\text{children}(\tau_2, \tau_1, i)| \leq |\text{ext}(\tau_2)|$$

Then $\text{onceOccurs}(\tau_1, \tau_2)$ and $\neg \text{moreOccurs}(\tau_1)$ prove that $\tau_1$ has the parent $\tau_2$. An XML-tree has less or equal children than $\tau_2$ nodes. The proof considers ways without cycle.

$$\text{way}(\tau_1, \tau_2) \quad \neg \text{cycle}(\tau_1) \quad |\text{ext}(\tau_1)| \leq |\text{ext}(\tau_2)|$$

An injective function proves the theorem choosing the next $\tau_2$ node of a $\tau_1$ node. The way proves with $T_5$ that a path exists. The function $f_4$ is proven injective. Otherwise there are nodes $v_1$ and $v_2$ with $v_3 = f_4(v_1) = f_4(v_2)$. With the paths to $v_3 T_2$ proves $v_1$ and $v_2$ are connected. Moreover, with $T_3$ a possible way from $\tau_1$ to itself is supplied. The contradiction with $\neg \text{cycle}(\tau_1)$ proves the inequation.

Finally, the induction proves $|\text{ext}(\tau_1)| \leq |\text{ext}(\tau_2)|$ with labels $\tau_1, \tau_2$ and $\tau_3$ that satisfy $\text{bounds}(\tau_1, \tau_3)$ and $\text{bounds}(\tau_3, \tau_2)$. The induction hypothesis proves $|\text{ext}(\tau_1)| \leq |\text{ext}(\tau_3)|$ and $|\text{ext}(\tau_3)| \leq |\text{ext}(\tau_2)|$. The next section presents a checker based on circular XML-specifications.

7 Deductive Checker

The previous section has proven that circular XML-specifications are unsatisfiable. This section presents a checker based on circular XML-specifications. The checker generates a representation with F-Logic [KLV95].
Objects represent elements and attributes, a class hierarchy represents the structural schema. The section presents a deductive checker based on circular XML-specifications. Section 6 proves the correctness of the checker that has been implemented with the DEAXS [His07] project. The checker generates F-Logic facts that are checked with Florid [HLS07].

The section presents the formalization of XML-specifications with F-Logic. Objects of class Element represent elements, the subclasses represent the normalized structural schemas [His07]. The checker is presented with the example defined in figure 3 and root teacher. The section formalizes the structural schema. Class

\[
\text{Element} \Rightarrow \text{Empty, Single, Text, Concat and Choice that represent the content models. A signature defines the class hierarchy and provides the method declaration.}
\]

\[
\text{Element}[\text{attributes} \Rightarrow \text{Attribute}].
\]

\[
\text{Empty} :: \text{Element}.
\]

\[
\text{Single} :: \text{Element}[\text{contents} \Rightarrow \text{Element}].
\]

\[
\text{Text} :: \text{Element}[\text{contents} \Rightarrow \text{Empty}].
\]

\[
\text{Concat} :: \text{Element}[\text{contents}@\text{integer} \Rightarrow \text{Element}].
\]

\[
\text{Choice} :: \text{Element}[\text{contents}@\text{integer} \Rightarrow \text{Element}].
\]

For example, teacher stores content model (teach, research). Object teacher is an instance of Concat. Attributes are stored with method attributes. Element teacher stores name, an instance of Attribute. The hierarchy is shown in Figure 5. Fact \( \tau : \text{Empty} \) represents \( \epsilon \). Contents \( P\tau = \tau_1 \) is represented with \( \tau : \text{Single}[\text{contents} \Rightarrow \tau_1] \), fact \( \tau : \text{Concat}[\text{contents}@\text{(1)} \Rightarrow \tau_1; \text{contents}@\text{(2)} \Rightarrow \tau_2] \) represents \((\tau_1, \tau_2)\).

The structural schema of the example is represented.

\[
\text{name} : \text{Attribute}.
\]

\[
\text{instructor} : \text{Attribute}.
\]

\[
\text{teacher} : \text{Concat}[\text{contents}@\text{(1)} \Rightarrow \text{teach}; \text{contents}@\text{(2)} \Rightarrow \text{research}; \text{attributes} \Rightarrow \text{name}].
\]

\[
\text{teach} : \text{Concat}[\text{contents}@\text{(1)} \Rightarrow \text{subject}; \text{contents}@\text{(2)} \Rightarrow \text{subject}; \text{attributes} \Rightarrow \{\}].
\]

\[
\text{research} : \text{Text}[\text{contents} \Rightarrow \text{S}; \text{attributes} \Rightarrow \{\}].
\]

\[
\text{subject} : \text{Text}[\text{contents} \Rightarrow \text{S}; \text{attributes} \Rightarrow \text{instructor}].
\]

\[
\text{S}[\text{attributes} \Rightarrow \{\}].
\]
Rules define relation \( \text{way} \) for formalizing circular XML-specifications in section 4.

\[
\text{way}(X_1, X_1) \leftarrow X_1 : \text{Element}.
\]

\[
\text{way}(X_1, X_2) \leftarrow X_1 : \text{Single[contents} \rightarrow X_3] \land \text{way}(X_3, X_2).
\]

\[
\text{way}(X_1, X_2) \leftarrow X_1 : \text{Concat[contents}@1 \rightarrow X_3] \land \text{way}(X_3, X_2).
\]

\[
\text{way}(X_1, X_2) \leftarrow X_1 : \text{Choice[contents}@1 \rightarrow X_3; \text{contents}@2 \rightarrow X_4] \land \text{way}(X_3, X_2) \land \text{way}(X_4, X_2).
\]

The example satisfies \( \text{way}(\text{subject}, \text{subject}) \), \( \text{way}(\text{teacher}, \text{subject}) \) and the way from \text{teacher} to \text{subject}. Next, a rule proves a branch.

\[
\text{branch}(X_1, X_2) \leftarrow \text{way}(X_1, X_3) \land X_3 : \text{Concat[contents}@1 \rightarrow X_4; \text{contents}@2 \rightarrow X_5] \land \text{way}(X_4, X_2) \land \text{way}(X_5, X_2).
\]

Relation \( \text{branch}(X_1, X_2) \) is defined with a way from \( X_1 \) to \( X_3 \) that represents the branch with ways to the descendant. Elements \text{teacher} and \text{teach} have a branch to \text{subject}. The example satisfies \( \text{branch}(\text{teach}, \text{subject}) \) and a branch from \text{teacher} to \text{subject}. The section formalizes cycles.

\[
X_1[@\text{occurs} \rightarrow X_2] \leftarrow X_2 : \text{Single[contents} \rightarrow X_1].
\]

\[
X_1[@\text{occurs} \rightarrow X_2] \leftarrow X_2 : \text{Element[contents}@1 \rightarrow X_1].
\]

Attribute \( \text{occurs} \) (\( \text{Element[occurs} \rightarrow \text{Element}) \) is defined with content models. Element \text{subject} satisfies \( \text{subject}@\text{occurs} \rightarrow \text{teach} \). Possible ways are formalized.

\[
\text{possibleWay}(X_1, X_2) \leftarrow X_2[@\text{occurs} \rightarrow X_1].
\]

\[
\text{possibleWay}(X_1, X_2) \leftarrow X_2[@\text{occurs} \rightarrow X_3] \land \text{possibleWay}(X_1, X_3).
\]

The example satisfies \( \text{possibleWay}(\text{teacher}, X) \) for \( X \in \{\text{research}, \text{subject}, \text{teach}\} \).

\[
\text{cycle}(X_1) \leftarrow \text{possibleWay}(X_1, X_1).
\]

A cycle with \( X_1 \) is proven when \( X_1 \) has a possible way to itself. The example doesn’t have cycles. The section formalizes elements that occur with more content models.

\[
\text{moreOccurs}(X_1) \leftarrow X_1[@\text{occurs} \rightarrow X_2] \land X_1[@\text{occurs} \rightarrow X_3] \land X_2 \neq X_3.
\]

A label \( X_1 \) that occurs in two content models satisfies \( \text{moreOccurs}(X_1) \).

\[
\text{onceOccurs}(X_1, X_2) \leftarrow X_2 : \text{Single[contents} \rightarrow X_1].
\]

\[
\text{onceOccurs}(X_1, X_2) \leftarrow X_2 : \text{Concat[contents}@1 \rightarrow X_1; \text{contents}@2 \rightarrow X_3] \land X_1 \neq X_3.
\]

\[
\text{onceOccurs}(X_1, X_2) \leftarrow X_2 : \text{Concat[contents}@1 \rightarrow X_3; \text{contents}@2 \rightarrow X_1] \land X_1 \neq X_3.
\]

Single content models \( \tau_1 \) and concatenations with \( \tau_3 \ (\tau_3 \neq \tau_1) \) that are stored for \( \tau_2 \) prove \( \text{onceOccurs}(\tau_1, \tau_2) \). The example satisfies \( \text{onceOccurs}(\text{research}, \text{teacher}) \). Next, \( \text{bounds} \) are defined.

\[
\text{bounds}(X_1, X_2) \leftarrow \text{anchor}(X_1, X_2).
\]

\[
\text{bounds}(X_1, X_2) \leftarrow \text{bounds}(X_1, X_3) \land \text{bounds}(X_3, X_2).
\]

\[
\text{bounds}(X_1, X_2) \leftarrow \text{onceOccurs}(X_1, X_2) \land \neg \text{moreOccurs}(X_1).
\]

\[
\text{bounds}(X_1, X_2) \leftarrow \text{way}(X_1, X_2) \land \neg \text{cycle}(X_1).
\]

An anchor bounds elements with integrity. XML-specifications with \( \tau_1[L_1] \rightarrow \tau_1 \) and \( \tau_1[L_1] \subseteq \tau_2[L_2] \) satisfy \( \text{anchor}(\tau_1, \tau_2) \). The example has an anchor. With the integrity constraints \( \text{subject.instructor} \rightarrow \text{subject} \) and \( \text{subject.instructor} \subseteq \text{teacher.name} \) the example proves \( \text{anchor}(\text{subject}, \text{teacher}) \). Element \( X_2 \) that bounds \( X_3 \) that bounds \( X_1 \) satisfies \( \text{bounds}(X_1, X_2) \). Elements that occur once satisfy \( \text{bounds} \). Ways from \( X_1 \) to \( X_2 \) without cycle satisfy \( \text{bounds}(X_1, X_2) \). The example proves \( \text{subject} \) with \( \text{teacher} \). Florid [HLS07] proves that the example is circular.

\[
\text{way}(\text{teacher}, \text{teacher}).
\]

\[
\text{branch}(\text{teacher}, \text{subject}).
\]

\[
\neg \text{cycle}(\text{teacher}).
\]

\[
\text{bounds}(\text{subject}, \text{teacher}).
\]
The rules prove a branch. The root element teacher has a way to teach with a branch to subject. The example doesn’t have a cycle with teacher that is bounded with subject.

\[ \neg \text{way}(r, X_1) \land \text{branch}(X_1, X_2) \land \neg \text{cycle}(X_1) \land \text{bounds}(X_2, X_1). \]

The example is proven circular with teacher for \( X_1 \), subject for \( X_2 \) and the root teacher. The section has presented a checker for XML-specifications. Section 6 has proven the correctness of the checker. The checker has been implemented with the DEAXS [His07] project.

8 Conclusion

The previous section has presented a deductive checker. The contribution concludes with an overview. An extensive formalization is developed with Isabelle [Pau94b]. Details are presented in [His07]. Circular XML-specifications are formalized with an inductive method [Pau94a]. Section 6 proves that circular XML-specifications are unsatisfiable. Section 7 presents a checker based on circular XML-specifications. XML-specifications are represented with F-Logic [KSW95]. The correctness of the checker is proven. The checker is implemented with the DEAXS [His07] project. The checker normalizes structural schemas, generates graphs and the representation of XML-specifications with F-Logic [KSW95] that is checked with Florid [HLS07].

References

Reprinted from the paper: Reasoning with Powerdomains in Isabelle/HOLCF

Brian Huffman
Portland State University
brianh@cs.pdx.edu

Abstract. This paper presents the first fully-mechanized formalization of powerdomains, implemented in the HOLCF logic of the Isabelle theorem prover. The powerdomain library provides an abstract view of powerdomains to the user, hiding the complicated implementation details. The library also provides proof automation, in the form of sets of rewrite rules for solving equalities and inequalities on powerdomains.

1 Introduction

Powerdomains are a domain-theoretic analog of powersets, which were designed for reasoning about the semantics of nondeterministic programs.[12] The use of powerdomains for reasoning about nondeterminism (and domain-theoretic denotational semantics in general) has declined in recent years, which I believe is primarily due to their perceived complexity. Compared to other more syntactic approaches to semantics, domain theory and powerdomains require a lot of mathematical sophistication to understand. This is a significant barrier for anyone who might want to use denotational semantics to reason about computation. It is my hope that the existence of good formalized libraries will remove that barrier to the use of domain theory for denotational semantics.

In this paper I attempt to demonstrate that powerdomains are a natural way for functional programmers to reason about nondeterministic programs. Using Haskell-style monadic code as a starting point, Section 2 motivates the definition of a powerdomain. Section 3 examines the three main varieties of powerdomains, and attempts to convey some intuitions about their structures and what each is good for. For readers wishing to use the powerdomain library, Section 4 documents all of the powerdomain operations provided by the library, as well as some of the lemmas and proof automation that is available. Section 5 describes the implementation of the powerdomain library; understanding this section is not necessary in order to use the library, and may be skipped on first reading.

This paper assumes some familiarity with Haskell. In particular, I expect the reader to know about monads, and the monad laws. I also assume that the reader is familiar with some of the basics of domain theory, which is traditionally used for reasoning denotationally about Haskell programs.[4] In particular, the reader should know about bottoms ($\bot$), complete partial orders ($\sqsubseteq$), limits of chains, continuous functions, and admissible predicates.

2 Nondeterminism monads

From a functional programmer’s perspective, a powerdomain can be thought of as simply a special kind of monad for nondeterminism. In addition to the standard monad operations \textit{return} and \textit{bind}, a powerdomain also provides a binary operation for making a nondeterministic choice. In Haskell syntax, we can specify a subclass of monads that have such a binary choice operator:[10]

\begin{verbatim}
class (Monad m) => MultiMonad m where
 (+|+) :: m a -> m a -> m a
\end{verbatim}

Haskell programmers often use the list monad to model nondeterministic computations; functions indicate multiple possible return values by enumerating them in a list. In this case, the list append operator (+++) fills the role of nondeterministic choice.
instance MultiMonad [] where
  xs ++ ys = xs ++ ys

The list monad has the great advantage of being executable: If you code up a nondeterministic algorithm in the list monad, you can just run it and see the results. However, for reasoning about nondeterministic algorithms, the list monad falls short in two important ways.

First, the list monad is not abstract enough: There are many different lists that represent the same set of possible return values. For example, consider a nondeterministic integer computation \( f \) with three possible outcomes: a return value of 3, a return value of 5, or divergence (i.e. a return value of \( \bot \)). The lists \([3,5,\bot]\) and \([5,3,\bot,\bot]\) both represent the value of \( f \) equally well; both represent the set \( \{3,5,\bot\} \). If divergence were not a possibility, then we could canonicalize the lists by sorting and removing duplicates—but obviously this does not work in general.

The second problem is that the list monad does not behave well in the presence of infinite or partial output. The problem originates with the definition of append: If \( xs \) is an infinite list, then \( xs ++ ys \) does not depend on \( ys \) at all. If \( ys \) includes some possible outcomes that do not already occur in \( xs \), then they get thrown away. Similarly, if \( xs \) is a partial list, like \( 1 : 2 : \bot \), then \( xs ++ ys \) also ignores its second argument.

This problem is demonstrated by the following recursive nondeterministic computation. Any integer greater than or equal to 2 should be a possible result. However, when interpreted in the list monad, only even integers are included. The problem is that since the denotation of \( f \) is an infinite list, the “return 1” is never reached.

\[
f :: (\text{MultiMonad} \ m) \Rightarrow m \text{ Int}
f = \text{do } x \leftarrow \text{return } 0 \text{ ++ f ++ return } 1 \text{ return } (x+2)
\]

Another possible nondeterminism monad for Haskell is the binary tree, whose definition is shown below. The binary tree monad solves the second problem that lists had: Unlike the list append operator, the \( \text{Node} \) constructor never ignores either of its arguments, even if the other is partial or infinite. However, the problem of multiple representations remains; in fact this problem is even worse than before. Since the choice operator for trees is a data constructor, it doesn’t satisfy any non-trivial equalities, while list append is at least associative.

\[
data \text{ Tree } a = \text{Leaf } a \mid \text{Node } (\text{Tree } a) (\text{Tree } a)
\]

\[
\text{instance Monad Tree where}
\text{return } x = \text{Leaf } x
\text{Leaf } x >>= f = f x
\text{Node } l r >>= f = \text{Node } (l >>= f) (r >>= f)
\]

\[
\text{instance MultiMonad Tree where}
\text{l ++ r = Node l r}
\]

For doing formal reasoning about nondeterministic computations, an ideal nondeterminism monad should satisfy all the axioms listed in Fig. 1. We will call a monad that satisfies all seven laws a powerdomain. Laws 1–3 are just the standard Haskell monad laws. Law 4 says that bind distributes over the choice operator, and laws 5–7 state that choice is associative, commutative and idempotent. The list monad satisfies laws 1–5, but not 6 or 7; the binary tree monad satisfies only 1–4. There is no obvious way to define a true powerdomain directly in Haskell, but in the next section we will see how to define powerdomains mathematically.

Note that in addition to the seven powerdomain laws, there is another implicit requirement for powerdomains: All of the operations must be monotone and continuous, i.e. they must respect the cpo structure of the types on which they operate. In Haskell, every definable function is automatically continuous by construction, while in Isabelle, the logic permits the definition of functions which are not necessarily continuous. Continuity is a concept defined in Isabelle/HOLCF, and it is necessary to prove that each function defined in the library is continuous.

46
1. return \( x \gg= f = f \ x \)
2. \( \text{xs} \gg= \text{return} = \text{xs} \)
3. \( (\text{xs} \gg= f) \gg= g = \text{xs} \gg= (\lambda x \to f \ x \gg= g) \)
4. \( (\text{xs} +|+ \text{ys}) \gg= f = (\text{xs} \gg= f) +|+ (\text{ys} \gg= f) \)
5. \( (\text{xs} +|+ \text{ys}) +|+ \text{zs} = \text{xs} +|+ (\text{ys} +|+ \text{zs}) \)
6. \( \text{xs} +|+ \text{ys} = \text{ys} +|+ \text{xs} \)
7. \( \text{xs} +|+ \text{xs} = \text{xs} \)

**Fig. 1.** The powerdomain laws

3 Powerdomains

There are multiple ways to define a powerdomain with operations that satisfy all of the desired laws. The three most common are known as the upper, lower, and convex powerdomains. These are also respectively known as the Smyth, Hoare, and Plotkin powerdomains. Historically, each variety is also associated with a musical symbol: sharp (♯) for upper, flat (♭) for lower, and natural (♮) for the convex powerdomain.

Before we dive into the details of the various powerdomains, first let us introduce some more notation. We will borrow the variable naming convention often used for lists in Haskell: For values of powerdomain types we use names like \( \text{xs} \), \( \text{ys} \), or \( \text{zs} \), while for the underlying elements we use names like \( x \), \( y \), or \( z \).

Also, we will consistently use set-style notation when talking about powerdomains. The singleton set syntax \( \{\} \) denotes the monadic return operator, “unit”; and the set union symbol (\( \cup \)) denotes the nondeterministic choice operator, “plus”. Also, we will use set enumerations like \( \{x, y, z\} \) as shorthand for \( \{x\} \cup \{y\} \cup \{z\} \).

When necessary, we will indicate a specific powerdomain by using the appropriate musical symbol as a superscript.

3.1 Convex powerdomain

For a given element domain \( \alpha \), the convex powerdomain \( P^\flat(\alpha) \) is the free continuous domain-algebra over the constructors \( \{-\}^\sharp \) and \( (\cup^\sharp) \), modulo the associativity, commutativity, and idempotence of \( (\cup^\sharp) \). (This construction is explained in [1, §6.1]) The convex powerdomain is “universal” in a category-theoretical sense, in that there is a unique mapping (preserving unit and plus) from the convex powerdomain into any other powerdomain.

Freeness means two things here. First, it says that the convex powerdomain consists only of values that can be built up from applications of unit and plus (i.e. the convex powerdomain has “no junk”). Secondly, freeness also means that no nontrivial equalities between terms should hold, except those required by the laws (i.e. the convex powerdomain has “no confusion”).

In the context of complete partial orders, the “no junk” property has a slightly different meaning than it does for ordinary inductive datatypes. As a cpo, the convex powerdomain includes values built from a finite number of constructor applications, plus additional values that result as limits of chains. Thus the convex powerdomain has an induction rule like the following:

\[
\text{adm}(P) \quad \forall x. \ P(\{x\}^\sharp) \quad \forall x. \ y. \ P(\langle x, y \rangle) \quad \to \quad P(x) \to \ P(y) \to \ P(x \cup^\sharp y)
\]

Admissibility of \( P \) means that for any chain of elements \( x_i \) such that \( P(x_i) \) holds for all \( i \), \( P \) must also hold for the limit \( \bigsqcup_i x_i \). This side condition reflects the fact that some values are only expressible as limits of chains—most induction rules in HOLCF have a similar admissibility side condition. (HOLCF can automatically prove admissibility for most inductive predicates used in practice.)

We still need to check that we can satisfy all of the powerdomain laws from Fig. 1. Laws 5–7 hold by construction. We can use laws 1 and 4 as defining equations for the bind operator. Finally, it is straightforward to prove laws 2 and 3 by induction.
Definition 1. We say that \( x \) is a member of \( xs \) if \( \{x\} \cup xs = xs \).

If \( xs \) represents a nondeterministic computation, and \( x \) is one of the possible results, then \( x \) must be a member of \( xs \). However, the set of members is not necessarily equal to the set of possible results. Not every conceivable set of results can be precisely represented in the convex powerdomain, as the following theorem implies.

Theorem 1. Let \( xs \) be a value in a convex powerdomain. Then the set of members of \( xs \) is convex-closed.

Proof. Let \( x \) and \( z \) be members of \( xs \), and let \( y \) be any value between \( x \) and \( z \), such that \( x \subseteq y \) and \( y \subseteq z \). We will show that \( y \) is a member of \( xs \).

1. From \( y \subseteq z \), we have \( \{y\} \uplus ^{\sharp} xs \subseteq \{z\} \uplus ^{\sharp} xs \), by monotonicity. Then since \( z \) is a member of \( xs \), we have \( \{z\} \uplus ^{\sharp} xs = xs \). Therefore \( \{y\} \uplus ^{\sharp} xs \subseteq xs \).
2. From \( x \subseteq y \), we have \( \{x\} \uplus ^{\sharp} xs \subseteq \{y\} \uplus ^{\sharp} xs \), by monotonicity. Then since \( x \) is a member of \( xs \), we have \( \{x\} \uplus ^{\sharp} xs = xs \). Therefore \( xs \subseteq \{y\} \uplus ^{\sharp} xs \).

By antisymmetry we have \( \{y\} \uplus ^{\sharp} xs = xs \), thus \( y \) is a member of \( xs \). \( \square \)

Theorem 1 says that the set of members of \( xs \) includes at least the convex closure of the set of possible return values. In practice, this means that sometimes nondeterministic computations with different sets of possible outcomes nevertheless have the same denotation in the convex powerdomain.

Consider the domain of lifted booleans, which contains three values: True, False, and \( \bot \). On top of this, we can construct the domain of pairs of booleans, which is ordered component-wise. Now imagine we have a nondeterministic computation \( f \) which has exactly two possible return values: either (True, False) or (\( \bot \), \( \bot \)). Next, define a computation \( g \) which additionally has a third possible return value of (True, \( \bot \)). Here is how we might specify \( f \) and \( g \) in Haskell:

\[
\begin{align*}
f, g & : (\text{MultiMonad} \ m) \Rightarrow m (\text{Bool, Bool}) \\
f & = \text{return} (\text{True, False}) \uplus ^{\sharp} \text{return} (\text{undefined, undefined}) \\
g & = \text{return} (\text{True, undefined}) \uplus ^{\sharp} f
\end{align*}
\]

If we model these computations using the convex powerdomain monad, then the denotation of \( f \) is \( \{(\text{True, False}), (\bot, \bot)\}^{\sharp} \), and the denotation of \( g \) is \( \{(\text{True, } \bot), (\text{True, False}), (\bot, \bot)\}^{\sharp} \). But according to Theorem 1, these values are actually equal—the convex powerdomain does not distinguish between the computations \( f \) and \( g \). In general, two computations will be identified if their respective sets of possible results have the same convex closure.

This convex closure thing may seem a little weird; why bother with all this, when we could just represent multiple result values using ordinary sets? The weirdness is a small price to pay for a significant bonus: Since powerdomains are cpos, and all the operations are continuous, that means that we can freely use powerdomains with general recursion—something you cannot do with ordinary powersets.

### 3.2 Upper powerdomain

The upper powerdomain \( \mathcal{P}^{\uparrow}(\alpha) \) can be defined in the same manner as the convex powerdomain, except we require \( \cup ^{\uparrow} \) to satisfy one extra law:

\[
xs \cup ^{\uparrow} ys \subseteq xs \tag{2}
\]

(Note that due to commutativity, the statement \( xs \cup ^{\uparrow} ys \subseteq ys \) is equivalent.) This law makes the upper powerdomain into a semilattice, where \( xs \cup ^{\uparrow} ys \) is the meet, or greatest lower bound, of \( xs \) and \( ys \).

Theorem 2. Let \( xs \) be a value in an upper powerdomain. Then the set of members of \( xs \) is upward-closed.
Proof. Let \( x \) be a member of \( xs \), and let \( y \) be any value such that \( x \subseteq y \). We will show that \( y \) is a member of \( xs \).

1. From the symmetric form of Eq. 2, we have \( \{y\}^\parallel \cup^\parallel xs \subseteq xs \).
2. From \( x \subseteq y \), we have \( \{x\}^\parallel \cup^\parallel xs \subseteq \{y\}^\parallel \cup^\parallel xs \), by monotonicity.

Then since \( x \) is a member of \( xs \), we have \( \{x\}^\parallel \cup^\parallel xs = xs \).

Therefore \( xs \subseteq \{y\}^\parallel \cup^\parallel xs \).

By antisymmetry we have \( \{y\}^\parallel \cup^\parallel xs = xs \), thus \( y \) is a member of \( xs \).

A consequence of this theorem is that if \( \perp \) is a member of \( xs \), then everything is a member of \( xs \). In other words, if a nondeterministic computation has any possibility of returning \( \perp \), then according to the upper powerdomain semantics, nothing else matters—it might as well always return \( \perp \). For this reason, the upper powerdomain is good for reasoning about total correctness: if \( \perp \) is not a member of \( xs \), then you can be sure that \( xs \) denotes a computation that has no possibility of nontermination.

3.3 Lower powerdomain

The lower powerdomain \( P^\flat(\alpha) \) can also be defined similarly, by adding a different extra law:

\[
xs \subseteq xs \cup^\parallel ys
\]

This law makes the upper powerdomain into a semilattice, where \( xs \cup^\parallel ys \) is the join, or least upper bound, of \( xs \) and \( ys \).

Theorem 3. Let \( xs \) be a value in a lower powerdomain. Then the set of members of \( xs \) is downward-closed.

Proof. Similar to the proof of Theorem 2.

An immediate consequence of this theorem is that in the lower powerdomain, \( \perp \) is a member of everything. Equivalently, \( \{\perp\}^\parallel \) is an identity for the \( (\cup^\parallel) \) operation. In terms of nondeterministic computations, this means that the lower powerdomain semantics ignores any nonterminating execution paths. In contrast to the upper powerdomain, the lower powerdomain is better for reasoning about partial correctness, where you want to verify that if a computation terminates, then its result will satisfy some property.

3.4 Visualizing powerdomains

To help convey an intuition for the structure of the various kinds of powerdomains, this section includes diagrams of the powerdomain orderings over a few different element types. Fig. 2 shows all three powerdomains over a small flat domain, like the lifted booleans. Fig. 3 extends this to a slightly larger flat domain. Fig. 4 extends this in a different way by adding a top value.

Looking at Figs. 2 and 3, some generalizations can be made about powerdomains over flat cpos. The ordering on the lower powerdomain of any flat cpo is isomorphic to the subset ordering on the corresponding powerset. Also note that the lower powerdomain of any finite cpo has a greatest element, which corresponds to the set including all possible return values. In contrast, the upper powerdomain is almost like the lower powerdomain flipped upside-down, except that the bottom element stays at the bottom; the other singleton sets are maximal in this ordering.

For the lifted two-element type, note that the convex powerdomain has the structure of the lower powerdomain embedded inside it, but with a new value (excluding \( \perp \)) added above each old value. The convex powerdomain of the lifted three-element type is not shown (due to its size) but it is related to the lower powerdomain in the same way.

The four-element lattice is interesting because due to its symmetry, it clearly illustrates the duality between the upper and lower powerdomains. The lower powerdomain is structured exactly like the upper powerdomain, but with the order reversed.
Fig. 2. Lifted two-element type, with upper, lower, and convex powerdomains

Fig. 3. Lifted three-element type, with upper and lower powerdomains

Fig. 4. Four-element lattice, with upper, lower, and convex powerdomains
4 HOLCF powerdomain library

This section describes the user-visible aspects of the HOLCF powerdomain library. The implementation defines three new type constructors, one for each of the three powerdomain varieties. Each type has \texttt{unit} and \texttt{plus} constructors, and a monadic \texttt{bind} operator. Each type also has \texttt{map} and \texttt{join} operators, defined in terms of \texttt{unit} and \texttt{bind} in the same manner as Haskell’s \texttt{liftM} and \texttt{join}. The full list of types and constants is shown in Fig. 5.

The functions \texttt{convex_to_lower} and \texttt{convex_to_upper} are the mappings guaranteed to exist by the universal property of the convex powerdomain; they preserve \texttt{unit} and \texttt{plus}. Note that instead of the full function space (\(\Rightarrow\)), all functions use the HOLCF continuous function space type (\(\Rightarrow\)), indicating that they are continuous functions.

\begin{verbatim}
typedef 'a upper_pd
upper_unit :: 'a \rightarrow 'a upper_pd
upper_plus :: 'a upper_pd \rightarrow 'a upper_pd \rightarrow 'a upper_pd
upper_bind :: 'a upper_pd \rightarrow ('a \rightarrow 'b upper_pd) \rightarrow 'b upper_pd
upper_map :: ('a \rightarrow 'b) \rightarrow 'a upper_pd \rightarrow 'b upper_pd
upper_join :: 'a upper_pd upper_pd \rightarrow 'a upper_pd

typedef 'a lower_pd
lower_unit :: 'a \rightarrow 'a lower_pd
lower_plus :: 'a lower_pd \rightarrow 'a lower_pd \rightarrow 'a lower_pd
lower_bind :: 'a lower_pd \rightarrow ('a \rightarrow 'b lower_pd) \rightarrow 'b lower_pd
lower_map :: ('a \rightarrow 'b) \rightarrow 'a lower_pd \rightarrow 'b lower_pd
lower_join :: 'a lower_pd lower_pd \rightarrow 'a lower_pd

typedef 'a convex_pd
convex_unit :: 'a \rightarrow 'a convex_pd
convex_plus :: 'a convex_pd \rightarrow 'a convex_pd \rightarrow 'a convex_pd
convex_bind :: 'a convex_pd \rightarrow ('a \rightarrow 'b convex_pd) \rightarrow 'b convex_pd
convex_map :: ('a \rightarrow 'b) \rightarrow 'a convex_pd \rightarrow 'b convex_pd
convex_join :: 'a convex_pd convex_pd \rightarrow 'a convex_pd
convex_to_upper :: 'a convex_pd \rightarrow 'a upper_pd
convex_to_lower :: 'a convex_pd \rightarrow 'a lower_pd
\end{verbatim}

\textbf{Fig. 5.} Powerdomain types and constants defined in HOLCF

For convenience, the library also provides set-style syntax for \texttt{unit} and \texttt{plus}, similar to the notation used in this paper. Along with the definitions of types and constants, the library provides a significant body of lemmas. Each powerdomain type has an induction rule in terms of \texttt{unit} and \texttt{plus}. Rules about injectivity, strictness, compactness, and ordering are provided for the constructors. Also, the functor and monad laws are provided as lemmas.

4.1 Bifinite type class

HOLCF uses Isabelle’s axiomatic type class mechanism [16] to represent different kinds of domains. The main axiomatic type classes in HOLCF are \texttt{cpo} (chain-complete partial orders) and \texttt{pcpo} (pointed cpos). Unfortunately, the powerdomain constructions do not work over arbitrary cpos; they need some additional structure. In order to formalize powerdomains in HOLCF, it was necessary to add a new axiomatic class \texttt{bifinite}, which is a subclass of \texttt{pcpo}. I will have more to say about class \texttt{bifinite} in Section 5.
As far as a user of the library is concerned, it does not matter how class \texttt{bifinite} is defined; the important thing is that it should be preserved by all of type constructors that the user works with. In the current version of Isabelle, instances are provided for all type constructors defined in the HOLCF library: continuous function space, Cartesian product, strict product, strict sum, lifted cpos, and all three varieties of powerdomains. Flat domains built from countable HOL types are instances of \texttt{bifinite} as well.

A known problem is that the current implementation of the domain package does not generate instances of class \texttt{bifinite} for new types. In the current version of HOLCF, if a user wants to use a domain package--defined type with powerdomains, it will be necessary to manually prove that the type is an instance of class \texttt{bifinite}. Updating the domain package to work with the \texttt{bifinite} class is planned as future work.

4.2 Automation

To facilitate reasoning with powerdomains, the library provides various sets of rewrite rules that are designed to work well together.

\textbf{ACI normalization.} Isabelle’s simplifier is set up to handle permutative rewrite rules. For any associative-commutative operator, there is a set of three permutative rewrite rules that can convert any expression built from the operator into a normal form (grouped to the right, with terms sorted according to some term-ordering).[2] Two of the AC rewrites are simply the associativity and commutativity rules. The third is the left-commutativity rule. For ACI rewriting, we need a total of five rules: the three AC rewrites, plus the idempotency rule, and also (analogous to left-commutativity) left-idempotency.

\begin{align*}
(xs \cup ys) \cup zs &= xs \cup (ys \cup zs) \\
yz &= y \\
yz (xs \cup zs) &= xs \cup (ys \cup zs) \\
zs &= z \\
zs (xs \cup ys) &= xs \cup ys
\end{align*}

(4)

Permutative rewriting using the ACI rules results in a normal form where expressions are nested to the right, and the terms are in sorted order, with no exact duplicates. In HOLCF, this normalization can be accomplished for the convex powerdomains by invoking (\texttt{simp add: convex_plus_aci}). Similarly, upper\_plus\_aci and lower\_plus\_aci may be used with upper and lower powerdomains, respectively.

\textbf{Solving inequalities.} A common subgoal in a proof might be to show that one powerdomain expression approximates another. For each variety of powerdomain, there is a set of rewrites that can be used to automatically reduce an inequality on powerdomains down to inequalities on the underlying type.

\begin{align*}
\{x\}^\sharp \subseteq \{y\}^\sharp &\iff x \subseteq y \\
xz &\iff (xz \subseteq ys) \land (xs \subseteq zs) \\
(yz) &\subseteq \{z\}^\sharp &\iff (yz) \subseteq \{z\}^\sharp \\
(xz) &\subseteq \{yz\}^\sharp &\iff (xz) \subseteq \{yz\}^\sharp
\end{align*}

(5)

\begin{align*}
\{x\}^\flat \subseteq \{y\}^\flat &\iff x \subseteq y \\
(xz) &\subseteq \{yz\}^\flat &\iff (xz) \subseteq \{yz\}^\flat \\
\{x\}^\flat &\subseteq (yz) &\iff (x) \subseteq (ys) \land \{x\} \subseteq \{ys\} \\
\{x\}^\flat &\subseteq \{yz\} &\iff (x) \subseteq \{yz\} \land \{x\} \subseteq \{zs\}
\end{align*}

(6)

\begin{align*}
\{x\}^\flat \subseteq \{y\}^\flat &\iff x \subseteq y \\
\{x\}^\flat &\subseteq \{yz\} &\iff (x) \subseteq \{yz\} \land \{x\} \subseteq \{zs\} \\
(xz) &\subseteq \{yz\} &\iff (xz) \subseteq \{ys\} \land \{zs\} \subseteq \{yz\}
\end{align*}

(7)
For the upper and lower powerdomains, each has a set of three rewrite rules that covers all cases of comparisons. For example, invoking \((\text{simp add: upper\_pd\_less\_simps})\) will rewrite \(\{x, y\} \sqsubseteq \{y, z\}\) to \(x \sqsubseteq z \lor y \sqsubseteq z\), using the rules in Eq. (5). Similarly, \((\text{simp add: lower\_pd\_less\_simps})\) uses the rules in Eq. (6) to simplify inequalities on lower powerdomains.

For the convex powerdomain, the three rules in Eq. (7) are incomplete: They do not cover the case of \((xs \cup^\oplus ys) \sqsubseteq (zs \cup^\oplus ws)\). To handle this case, we will take advantage of the coercions from the convex powerdomain to the upper and lower powerdomains, along with the following ordering property:

\[xs \sqsubseteq ys \iff \text{to\_upper}(xs) \sqsubseteq \text{to\_upper}(ys) \land \text{to\_lower}(xs) \sqsubseteq \text{to\_lower}(ys)\]  

(8)

The rule set \textit{convex\_pd\_less\_simps} includes all rules from Eqs. (5)–(7), and a suitably instantiated Eq. (8) to cover the missing case.

\section{Using inequalities to solve non-trivial equalities.}

The ACI rewriting can take care of many equalities between powerdomain expressions, but the inequality rules can actually solve more. For example, using the assumptions \(x \sqsubseteq y\) and \(y \sqsubseteq z\), we will prove that \(\{x, y, z\} \sqsubseteq \{x, z\}\). By antisymmetry, we can rewrite this to the conjunction \((\{x, y, z\} \sqsubseteq \{x, z\}) \land (\{x, z\} \sqsubseteq \{x, y, z\})\). Next, we can use the method \((\text{simp add: convex\_pd\_less\_simps})\), and this subgoal reduces to \((y \sqsubseteq x \lor y \sqsubseteq z) \land (x \sqsubseteq y \lor z \sqsubseteq y)\). Finally, this is easily discharged using the assumptions \(x \sqsubseteq y\) and \(y \sqsubseteq z\).

\section{Implementation}

The development of powerdomains in HOLCF follows the ideal completion construction presented by Gunter and Scott in \cite{5}. Some alternative constructions are also given by Abramsky and Jung in \cite{1}; the ideal completion method was chosen because it required the formalization of a minimal amount of supporting theories, and it offered good opportunities for proof reuse.

\subsection{Class of bifinite domains}

The powerdomain construction used in HOLCF makes use of an alternative representation of domains, where we just consider the set of compact (i.e. finite) values, rather than the whole domain.\cite{1, 2.2.6} For this representation to work, we restrict our attention to algebraic cpos, where every value can be expressed as the limit of its compact approximants. This means that in an algebraic cpo the set of compact elements, together with the domain ordering on them, fully represents the entire domain. We say that the set of compact elements forms a basis for the domain, and the entire domain is a completion of the basis.

Most of HOLCF has been designed using the type class \textit{pcpo} of pointed complete partial orders. However, \textit{pcpo} types are not algebraic in general, and the ideal completion construction only works with algebraic cpos. Therefore it was necessary to add a new type class to HOLCF.

The class \textit{bifinite} is defined as follows. It fixes a sequence of functions \textit{approx}, and assumes four class axioms:

1. The \textit{approx} form a chain
2. The least upper bound \((\bigsqcup_n \text{approx})\) is the identity function
3. Each \textit{approx} is idempotent
4. Each \textit{approx} has finite range

The HOLCF \textit{bifinite} class actually corresponds to the “\(\omega\)-bifinite” domains, which have a countable basis; the usual definition of “bifinite” \cite{1, 4.2} has no such restriction, and would be equivalent to allowing any directed set of approx functions, rather than a countable basis. Bifinite domains were originally defined by Plotkin as limits of expanding sequences of finite posets, who used the name “SFP domain”.\cite{12}

Of all the various classes of domains to choose from, the definition of \textit{bifinite} was chosen for the following reasons:
All bifinite types are algebraic: Every bifinite type has a countable basis of compact elements, given by the union of the ranges of the approx functions.

In bifinite types, every directed set contains a chain with the same limit. This means that in class bifinite, the notions of directed-continuity and chain-continuity coincide. This is important for fitting the ideal completion construction (which uses directed sets) into HOLCF (which defines everything with chains).

The bifinite class is closed under all type constructors used in HOLCF, including the convex powerdomain.

5.2 Ideal completion

Given a basis \( \langle B, \preceq \rangle \), we can reconstruct the full algebraic cpo. The standard process for doing this is called ideal completion, and it is done by considering the set of ideals over the basis:

**Definition 2.** A set \( S \) is an ideal with respect to partial preorder relation \( (\preceq) \) if it has the following properties:

- \( S \) is nonempty: \( \exists x. x \in S \)
- \( S \) is downward-closed: \( \forall x y. x \preceq y \rightarrow y \in S \rightarrow x \in S \)
- \( S \) is directed (i.e. has an upper bound for any pair of elements):
  \[ \forall x y. x \in S \rightarrow y \in S \rightarrow (\exists z. z \in S \land x \preceq z \land y \preceq z) \]

A principal ideal is an ideal of the form \( \{ y. y \preceq x \} \) for some \( x \), denoted \( \downarrow x \).

The set of all ideals over \( \langle B, \preceq \rangle \) is denoted \( \text{Idl}(B) \); when ordered by subset inclusion, \( \text{Idl}(B) \) forms an algebraic cpo. The compact elements of \( \text{Idl}(B) \) are exactly those represented by principal ideals.

Note that the relation \( (\preceq) \) does not need to be antisymmetric. For \( x \) and \( y \) that are equivalent (that is, both \( x \preceq y \) and \( y \preceq x \)) the principal ideals \( \downarrow x \) and \( \downarrow y \) are equal. This means that the ideal completion construction automatically takes care of quotienting by the equivalence induced by \( (\preceq) \).

The ideal completion construction is formalized in HOLCF using Isabelle’s locale mechanism.[8] The library defines a locale preorder that fixes a type corresponding to the basis \( B \), and a preorder relation on that type; within this locale, a predicate ideal is defined. Within the preorder locale, the main lemma proved is that the union of a chain of ideals is itself an ideal—which shows that the ideal completion is a cpo.

All three of the powerdomains in the library are defined by ideal completion. For an basis, the library defines a type \( 'a \text{ pd\_basis} \), which consists of nonempty, finite sets of compact elements of type \( 'a \). Following [5, §5.2], each of the three powerdomains is defined as an ideal completion over the same basis, but each uses a different preorder relation:

\[
\begin{align*}
a \preceq^\oplus b & \iff \forall x \in a. \exists y \in b. x \sqsubseteq y \\
a \preceq^\otimes b & \iff \forall y \in b. \exists x \in a. x \sqsubseteq y \\
a \preceq^\vee b & \iff a \preceq^\oplus b \land a \preceq^\otimes b
\end{align*}
\] (9)

5.3 Continuous extensions of functions

A continuous function on an algebraic cpo is completely determined by its action on compact elements. This suggests a method for defining continuous functions over ideal completions: First, define a function from the basis \( B \) to a cpo \( C \) such that \( f \) is monotone, i.e. \( x \preceq y \) implies \( f(x) \sqsubseteq f(y) \). Then there exists a unique function \( \hat{f} : \text{Idl}(B) \rightarrow C \) that agrees with \( f \) on principal ideals, i.e. for all \( x \), \( \hat{f}(\downarrow x) = f(x) \). We say that \( \hat{f} \) is the continuous extension of \( f \).

On top of the preorder locale, HOLCF defines another locale ideal\_completion which fixes a second type corresponding to \( \text{Idl}(B) \). It also fixes a function principal of type \( B \rightarrow \text{Idl}(B) \). Within this locale, a
function \texttt{basis\_fun} is defined, which takes a monotone function \( f \) as an argument, and returns the continuous extension \( f \).

The continuous extension is defined by mapping the function \( f \) over the input ideal, and then taking the least upper bound of the resulting directed set: \( f(S) = \bigcup_{x \in S} f(x) \). Ordinarily, the result type \( C \) would need to be a directed-complete partial order to ensure that this least upper bound exists; however, the HOLCF library uses a different method which allows \( C \) to be any chain-complete partial order.

HOLCF defines a third locale \texttt{basis\_take}, which fixes a chain of \texttt{take} functions over the basis elements—it is like a version of the \texttt{bifinite} class for bases. The \texttt{basis\_take} locale ensures that the ideal completion \( \text{Idl}(B) \) is a bifinite domain. It is also used with the definition of \texttt{basis\_fun} to construct a chain with the same limit as the directed set \( \bigcup_{x \in S} f(x) \), which allows \( C \) to be an arbitrary chain-cpo.

The \texttt{basis\_fun} combinator is used to define the powerdomain constructors \texttt{unit} and \texttt{plus} in terms of the singleton and union operations on the \texttt{pd\_basis} type. The \texttt{bind} operators are also defined using \texttt{basis\_fun}, in terms of a finite-set fold operation on \texttt{pd\_basis}. Finally, to prove the \texttt{bifinite} class instance, the approx functions are also defined with \texttt{basis\_fun}, in terms of the \texttt{take} functions on \texttt{pd\_basis}.

### 5.4 Transferring properties to the completed domain

Once the powerdomain types are defined using ideal completion, with operations defined by continuous extension, the final step is to prove the relevant lemmas. For example, consider the lower powerdomain law \( xs \sqsubseteq xs \cup ys \). In the case where \( xs \) and \( ys \) are both compact (i.e. represented by principal ideals) the proof follows easily from the definitions. Since \( xs \sqsubseteq xs \cup ys \) is an admissible predicate on both \( xs \) and \( ys \), this is in fact sufficient to show that it holds for all \( xs \) and \( ys \).

Other properties are more tricky to transfer. For example, consider the rule \( \{x\}^2 \sqsubseteq \{y\}^2 \implies x \sqsubseteq y \). As before, this property is easy to prove for compact \( x \) and \( y \). However, we cannot immediately infer that it holds for all \( x \) and \( y \), since (because of the implication) this is not an admissible predicate.

The proof of \( \{x\}^2 \sqsubseteq \{y\}^2 \implies x \sqsubseteq y \) requires a few extra steps, making use of the approx functions from the \texttt{bifinite} class: To prove \( x \sqsubseteq y \), it will be sufficient to show that for all \( n \), \( \approx_n x \sqsubseteq \approx_n y \). Now, from \( \{x\}^2 \sqsubseteq \{y\}^2 \) we have \( \approx_n \{x\}^2 \sqsubseteq \approx_n \{y\}^2 \), by monotonicity; then from the definition of approx on the convex powerdomain, this simplifies to \( \approx_n x \sqsubseteq \approx_n y \).

Finally, since \( \approx_n x \) and \( \approx_n y \) are compact, we can easily show that \( \approx_n x \sqsubseteq \approx_n y \). All of the rules listed in Eqs. (5)–(8) use a similar proof.

### 6 Related work

There are several theorem prover formalizations of domain theory in existence. The current development is built on top of HOLCF, originally implemented by Regensburger, and later extended by many others.\cite{13,9} HOLCF does not formalize very many different classes of domains; most concepts are defined in terms of pointed chain-complete partial orders and chain-continuity, which is the minimum amount of structure required to define a fixed-point combinator. It is intended to be used as a library for users to define datatypes and recursive functions and algorithms on them.

In the mid-1990s a group from the University of Ulm formalized parts of domain theory in PVS.\cite{3} Its design goals appear to be similar to HOLCF—it includes just enough of domain theory to formalize fixed-points and fixed-point induction.

A formalization of domain theory with rather different goals is “Elements of Domain Theory”, implemented in Coq in the 1990s by Kahn. It is based on the definitions and lemmas from \cite{7}. This development defines several classes of domains, including directed-complete partial orders, omega-algebraic cpos, and bounded-complete domains. However, it does not define any type constructors. In contrast to HOLCF, it does not appear to be application-oriented; it seems the main intent was to formalize the textbook-style definitions and lemmas from the paper.

Other formalizations use a different logic to ensure that all functions are continuous by construction, such as the LCF system by Paulson.\cite{11} Another interesting approach is taken by Reus with his development of
synthetic domain theory in LEGO.[14] Instead of defining classes of domains in terms of a domain ordering, it starts by introducing a subobject classifier, which is characterized by a collection of axioms. The soundness of the construction is justified by a separate model.

Relevant uses of powerdomains include modeling interleaved and parallel computation. Papaspyrou uses the convex powerdomain, together with the state and resumption monad transformers, to model impure languages with unspecified evaluation order.[10] Along similar lines, Thiemann used a type of state monad built on top of powerdomains to reason about concurrent computations.[15] The monad transformers used in these works, specifically the resumption monad transformer, have been studied in HOLCF by Huffman, et al.[6]

7 Conclusion and future work

The powerdomain library described here is included as part of Isabelle2008 theorem prover. It can already be used to prove properties of simple nondeterministic algorithms, with automation for certain kinds of subgoals. Future work will focus on better integration with the HOLCF domain package: Bifinite class instances must be generated for all new datatypes. Also, the domain package needs to be extended to allow recursive type definitions involving powerdomains—this will enable the use of powerdomains for modeling parallel computation and concurrency.

References

Abstract. We show how the dependency pair approach, commonly used to modularize termination proofs of rewrite systems, can be adapted to establish termination of recursive functions in a system like Isabelle/HOL or Coq. It turns out that all that is required are two simple lemmas about wellfoundedness.

1 Introduction

Termination proofs are essential in theorem proving, as they are required to justify the definition of recursive functions.

Dependency pairs [1,8] are currently one of the most successful approaches to prove termination of term rewrite systems (TRSs). They have been successfully adapted to other situations like functional programs [7]. So it seems natural to try to use such techniques in a theorem proving context.

However, there are two main obstacles that make an adoption difficult: First, formalizing the underlying theory is a major technical effort, since the proofs are often quite technical and non-trivial, and involve a large variety of specialized concepts.

Second, there is a significant gap between the very syntax-centric view of term rewriting (“prove that a given list of rewrite rules over a certain signature allows no infinite rewrite sequences”) and the more semantic view that we need here (“prove that a given relation over, say, natural numbers is wellfounded”), where terms, signatures and reductions never occur on the object level.

The first obstacle has been attacked by two recent formalization efforts [2,5], where large parts of the underlying metatheory were formally verified. This approach allows the certification of TRS proofs in an interactive proof assistant, but it cannot be used to justify the definition of recursive functions yet.

In this paper we are concerned with the second obstacle. We take the slightly more abstract relational view that is appropriate to deal with the termination proof obligations arising from function definitions. It turns out that some technical issues disappear in this setting, and that dependency pair proofs are nothing more than a clever application of the following two lemmas:

Lemma 1. $\text{wf } R \implies \text{wf } S \implies R \circ S \subseteq R \implies \text{wf } (R \cup S)$

Lemma 2. $\text{wf } (R \cup S) = \text{wf } ((R \cup S) \circ R \cup S)$

This paper is structured as follows: After introducing some basic notions and notation in §2, we revisit briefly the traditional dependency pair approach in §3. In §4, we explain dependency pair proofs in the setting of Isabelle/HOL, and give some simple examples in §5 and §6.

2 Preliminaries

We work in Isabelle/HOL [12], but as we do not rely on any special features, the same ideas also apply in a system with different foundations, such as Coq. All theorems and proofs presented in this paper were mechanically checked by Isabelle.
2.1 Notation

In our notation, relations are represented as sets of pairs, and composition is defined as

\[ R \circ S = \{ (x, z) \mid \exists y. (x, y) \in R \land (y, z) \in S \} . \]

We will frequently write comprehensions of the form \( \{ f x y \mid x y. P x y \} \), which abbreviates \( \{ u. \exists x y. u = f x y \land P x y \} \).

2.2 Termination Proof Obligations

Consider the following example function:

\[
\begin{align*}
\text{foo } n \ 0 &= n \\
\text{foo } 0 \ (\text{Suc } m) &= \text{foo } (\text{Suc } m) \ (\text{Suc } m) \\
\text{foo } (\text{Suc } n) \ (\text{Suc } m) &= \text{foo } n \ m
\end{align*}
\]

When a recursive function is defined in Isabelle, the system automatically produces a proof obligation that corresponds to the termination of the function (cf. [10,13]). This proof obligation states the wellfoundedness of the function’s call relation (which relates each possible argument of the function with the resulting recursive calls). For the above definition, we obtain the goal

\[
\text{wf } \{(\text{Suc } m, \text{Suc } m), (0, \text{Suc } m) \mid m. \text{True} \} \cup
\{(n, m), (\text{Suc } n, \text{Suc } m) \mid m n. \text{True} \}
\]

Note how each recursive call is written as a relation comprehension. Isabelle’s standard definition of wellfoundedness follows the convention that the “smaller” element in a relation appears on the left. Consequently, the arguments from the left hand sides of the equations above must go to the right and vice-versa.

In previous work (e.g. Bulwahn et al. [3]), this proof obligation was sometimes presented differently, asking for an embedding into another wellfounded relation:

1. \( \text{wf } ?R \)
2. \( \forall m. ((\text{Suc } m, \text{Suc } m), (0, \text{Suc } m)) \in ?R \)
3. \( \forall n m. ((n, m), (\text{Suc } n, \text{Suc } m)) \in ?R \)

In this paper, we prefer the first version, which is equivalent, but gives us more flexibility to modify the problem. It is easy to convert between one form of the goal to the other.

The general form of the proof obligation is the following:

\[
\text{wf } \{ \{ (r_1, l_1) \mid v_1 \ldots v_{m_1}, \Gamma_1 \} \}
\cup \ldots
\cup \ldots
\cup \{ (r_n, l_n) \mid v_1 \ldots v_{m_n}, \Gamma_n \}
\]

Thus we must prove wellfoundedness of a relation that is given as a union of relation comprehensions, each reflecting a recursive call from \( l_k \) to \( r_k \) under the conditions \( \Gamma_k \). Here, the bound variables \( v_1, \ldots, v_m \) can occur in \( r, l \) and \( \Gamma \).

If we take the relations \( C \) and \( D \) corresponding to two recursive calls, we can form their composition \( D \circ C \), which expresses the call \( C \) which is immediately followed by \( D \). Such a composition can again be written as a comprehension:

\[
\begin{align*}
\{ (a, b x) \mid x. \ P x \} \circ \{ (c y, d y) \mid y. \ Q y \} &= \\
\{ (a, d y) \mid x y. \ b x = c y \land P x \land Q y \}
\end{align*}
\]

\(^1\) It is a bit counterintuitive that the composition is written in the “wrong” order. Again, this comes from the fact that the smaller element is written left.
3 Dependency Pairs in Rewriting

We give a very brief (and slightly oversimplified) introduction to dependency pairs as they are used in termination proofs for term rewriting. The readers who are interested in more details should consult the original literature [1,8].

We consider first order terms over a fixed signature. Given a term rewrite system $R$, i.e. a set of rules of the form $l \rightarrow r$, we say that a function symbol is defined, if it occurs in root position on the left hand side of some rule in $R$. The set of dependency pairs (DPs) of $R$ is defined as

$$DP(R) = \{ t^\sharp \rightarrow t^\sharp \mid l \rightarrow r \in R, t \text{ is a subterm of } r \text{ with a defined root symbol} \}$$

Here, $t^\sharp$ is just the term $t$ whose root symbol is marked with a $\sharp$, to distinguish it from a normal function symbol. Thus, dependency pairs just capture the usual notion of a recursive call.

A (finite or infinite) sequence of dependency pairs $s_1 \rightarrow t_1, s_2 \rightarrow t_2, \ldots$ is called a chain, if there is a substitution $\sigma$, such that $\sigma(t_i) \rightarrow^*_R \sigma(s_{i+1})$ for all $i$. Such chains model sequences of recursive calls as they can occur during a reduction. A TRS terminates if and only if there are no infinite chains.

The dependency graph of a TRS $R$ is the graph with the nodes $DP(R)$ and an edge between two DPs, if they form a chain (of length two). The dependency graph serves as a simple model of control flow and allows the decomposition of a termination problem into smaller parts: if it has more than one strongly connected component (SCC), these can be split apart and dealt with separately.

The resulting smaller problems can be tackled using orderings: for a pair of relations $(\prec, \preceq)$ with certain properties\(^2\) (called a reduction pair), we can remove all DPs that respect $\prec$, if the remaining DPs respect $\preceq$. Note that after removing DPs like this, the dependency graph may again fall apart into different SCCs. We continue with the repeated decomposition and application of reduction pairs until we reach the empty set of DPs.

It is undecidable in general whether two DPs form a chain. So the dependency graph is usually approximated using some safe heuristic\(^1\). The most commonly used heuristic simply checks if the right hand side of one DP and the left hand side of the other DP are unifiable after replacing all defined symbols by fresh variables and renaming the variables apart.

Example We now prove termination of `foo` using dependency pairs. The dependency pairs are the two recursive calls, and the dependency graph looks like this:

$$f^\sharp(0, \text{Suc } m) \rightarrow f^\sharp(\text{Suc } m, \text{Suc } m) \quad \text{if } f^\sharp(\text{Suc } n, \text{Suc } m) \rightarrow f^\sharp(n, m)$$

There is only one SCC, so we cannot split the problem into pieces. Now, we note that the second argument is decreasing in the second call, and it stays the same in the first one\(^3\). This allows us to remove the second call, and just one dependency pair remains:

$$f^\sharp(0, \text{Suc } m) \rightarrow f^\sharp(\text{Suc } m, \text{Suc } m)$$

Since there is no edge here, there cannot possibly be a loop, and we are done. This step is remarkable, since we do not need to argue about decreasing measures and the like. It is enough to know that the call cannot happen twice in a row.

Dependency pair proofs do not yield a simple characterization of a well-founded order compatible with all recursive calls. However, this is not required, as we will see in the next section.

\(^2\) In addition to $\prec$ being wellfounded, they must be closed under substitutions, $\preceq$ must be closed under contexts and $\prec \circ \preceq \subseteq \prec$ and $\preceq \circ \succ \subseteq \succ$. Our adapted version in §4 will drop some of these requirements.

\(^3\) In rewriting this is formalized with a subterm ordering and a so-called argument filtering that selects the second argument.
4 Shallow Dependency Pairs in HOL

We now see how we can “mimic” dependency pair proofs in a very shallow way, to solve proof obligations of the type given in §2. The relation comprehensions for the different recursive calls will take the place of the dependency pairs.

Before we start, note a trivial consequence of Lemma 2 (with $S = \emptyset$):

**Corollary 1.** $\text{wf } R = \text{wf } (R \circ R)$

4.1 Dependency Graph

Our dependency graph has an edge from call $C$ to call $D$ (each represented by a relation), iff $C$ can be followed by $D$, i.e. iff

$D \circ C \neq \emptyset$

Like in §3, this property is undecidable, but we can safely approximate the dependency graph by drawing an edge whenever we are unable to find an proof of $C \circ D = \emptyset$ using some automated tactic we have at hand.

Whenever we have more than one strongly connected component, we can apply a divide-and-conquer strategy: Let $R$ and $S$ be sets of calls, such that no call from $R$ can happen after some call from $S$. This means that $R \circ S = \emptyset \subseteq R$ holds, and we can apply Lemma 1 to split the two sets of calls into independent sub-problems. This step can be repeated until the graph is split into its SCCs.

4.2 Removing trivial SCCs

With the decomposition steps above, we might finally end up with the trivial SCC that just consists of a single call $C$ that is not reachable from itself. This corresponds to a graph with one node and no edges.

Formally, since there is no edge, we have $C \circ C = \emptyset$. Using Corollary 1, we can rewrite the goal to $\text{wf } (C \circ C)$, which is trivial, since the empty set is always wellfounded.

4.3 Using reduction pairs

When problem cannot be simplified any more, some real progress needs to be made. If one of the calls strictly decreases with respect to a certain ordering, and all the other calls are weakly decreasing (“less or equal”), then that call can be removed.

The notion of strict and weak decrease wrt. arbitrary orderings is expressed by a reduction pair, that is a pair of relations $R_<$ and $R_\leq$ satisfying:

reduction-pair $(R_<, R_\leq) = (\text{wf } R_\leq \land R_\leq \circ R_\leq \subseteq R_\leq)$

The canonical example of a reduction pair is a quasi-order, whose strict part is wellfounded, but the above definition is more general, since transitivity is not required. We usually construct reduction pairs from measure functions into the natural numbers by taking the inverse images of $<$ and $\leq$.

The following is a direct consequence of Lemma 1 and the above definition:

**Corollary 2.** reduction-pair $(R_<, R_\leq) \Rightarrow S \subseteq R_\leq \Rightarrow S' \subseteq R_\leq \Rightarrow \text{wf } S' \Rightarrow \text{wf } (S \cup S')$

Operationally, we are allowed to remove some calls $S$ from a proof obligation, if we can find a reduction pair $(R_<, R_\leq)$, such that $S$ can be embedded into $R_<$ and $S'$ can be embedded into $R_\leq$.

Of course, finding useful reduction pairs can be very difficult in general, but in practice many useful relations can be found by very simple heuristics. Basically, we can use the same heuristics as in [3].

Note that after removing a call, the dependency graph may again fall apart into different SCCs.
5 Examples

Given these tools, the termination proof for foo is easy. We start with the goal we already saw in §2:

\[
\begin{align*}
\text{wf } &\{(\text{(Suc } m, \text{ Suc } m), (0, \text{ Suc } m)) \text{ } | \text{ } m. \text{ True}\} \cup \\
&\{(\text{(n, m), (Suc } n, \text{ Suc } m)) \text{ } | m \text{ n. True}\}
\end{align*}
\]

First, we can attack the second call using the reduction pair given by the measure function \( \text{snd} \). Proving strict and weak descent of the second and the first call, respectively, is simple. Hence, applying Corollary 2, we get rid of that call, and are left with just the goal

1. \( \text{wf } \{(\text{(Suc } m, \text{ Suc } m), (0, \text{ Suc } m)) \text{ } | \text{ } m. \text{ True}\} \)

This SCC is trivial, since we can easily show that \( C \circ C = \emptyset \) for this call, which concludes the proof.

Let us look at another (artificial) example. The function \( \text{bar} \) has two modes. Depending on a boolean flag, either the first or the second argument decreases:

\[
\begin{align*}
\text{bar True } (\text{Suc } n) &\text{ } m = \text{bar True } n (\text{Suc } m) \\
\text{bar True } 0 &\text{ } m = \text{bar False } 0 \text{ } m \\
\text{bar False } n (\text{Suc } m) &\text{ } = \text{bar False } (\text{Suc } n) \text{ } m \\
\text{bar False } n \text{ } 0 &\text{ } n
\end{align*}
\]

This specific control flow becomes clearly visible in the dependency graph:

\[
\begin{align*}
\text{True, Suc } n, \text{ } m &\rightarrow (\text{True, Suc } m) \\
\text{True, } 0, \text{ } m &\rightarrow (\text{False, } 0, \text{ } m) \\
\text{False, } n, \text{ } \text{Suc } m &\rightarrow (\text{False, Suc } m, \text{ } n)
\end{align*}
\]

Now, we apply the above techniques to prove termination. As usual, we start with the goal:

1. \( \text{wf } \{(\text{(True, n, Suc } m), (\text{True, Suc } n, \text{ m)) } | n \text{ m. True}\} \cup \\
\{(\text{False, 0, m), (True, 0, m)) } | m. \text{ True}\} \cup \\
\{(\text{False, Suc } n, m, (\text{False, } n, \text{ Suc } m)) \text{ } | n \text{ m. True}\})

Using Lemma 1, we can split off the first call, since it can never follow one of the other calls:

1. \( \text{wf } \{(\text{True, n, Suc } m), (\text{True, Suc } n, m)) \text{ } | n \text{ m. True}\}
2. \( \text{wf } \{(\text{False, 0, m), (True, 0, m)) } | m. \text{ True}\} \cup \\
\{(\text{False, Suc } n, m, (\text{False, } n, \text{ Suc } m)) \text{ } | n \text{ m. True}\})

The other two calls can be split in the same way, and we have three independent sub-problems:

1. \( \text{wf } \{(\text{True, n, Suc } m), (\text{True, Suc } n, m)) \text{ } | n \text{ m. True}\}
2. \( \text{wf } \{(\text{False, 0, m), (True, 0, m)) } | m. \text{ True}\}
3. \( \text{wf } \{(\text{False, Suc } n, m, (\text{False, } n, \text{ Suc } m)) \text{ } | n \text{ m. True}\}

Now that we have isolated them, each call can be solved on its own, and each in a different way: The first call is decreasing in the second argument, the second one is trivial, by Corollary 1, and the third one again has a measure, this time the third argument.
6 Merging

Sometimes, recursive calls occur on arguments that are not decreasing with respect to any obvious relation. But the increase might only be temporary. These functions are usually difficult to handle.

The following example occurs in a reflected arithmetic decision procedure formalized by Chaieb [4]. It operates on a datatype representing numeric expressions:

```plaintext
datatype num = C int | Bound nat | CN nat int num | Neg num | Add num num
          | Sub num num | Mul int num | Floor num | CF int num num
```

The function `simpnum` simplifies `num` expressions, using the helper functions `numneg`, `numadd`, whose definitions we omit. Here is the equations for `simpnum`:

```plaintext
simpnum (C j) = C j
simpnum (Bound n) = CN n 1 (C 0)
simpnum (Neg t) = numneg (simpnum t)
simpnum (Add t s) = numadd (simpnum t) (simpnum s)
simpnum (Sub t s) = numsub (simpnum t) (simpnum s)
simpnum (Mul i t) = (if i = 0 then C 0 else nummul (simpnum t) i)
simpnum (Floor t) = numfloor (simpnum t)
simpnum (CN n c t) = (if c = 0 then simpnum t else CN n c (simpnum t))
simpnum (CF c t s) = simpnum (Add (Mul c (Floor t)) s)
```

Now consider the last equation: Here obviously the argument gets larger, at least if we use the usual size measure, which just counts constructors. However this increase is just temporary, and after a few more evaluation steps, we will end up again with an argument of smaller size. Put differently, if we unfolded the recursive call in the last equation a few times, we would get

```plaintext
simpnum (CF c t s) =
numadd (if c = 0 then C 0 else nummul (numfloor (simpnum t)) c) (simpnum s)
```

where all calls are again structurally decreasing.

In rewriting, this simply corresponds to the rewriting of the right hand sides of the dependency pairs [8, Thm. 21]. The same idea has also been used in an extended termination checker [11] for ACL2 [9], where it is called merging, but only justified metatheoretically.

In our framework, we can give a formal justification of this step, using relation composition: Lemma 2 allows us to replace some call `R` in a goal `wf (R ∪ S)` by `(R ∪ S) ◦ R` Informally, this means that we compose the `R`-step with all its possible successors. Since the composition of two relation comprehensions again has the form of a relation comprehension, we again obtain a subgoal of the form that can be handled by the methods described above.

In the `simpnum` example, `R` would be the problematic call, and `S` the union of the other calls. After the merging, most resulting calls can be simplified away, since after the `CN` call there must follow an `Add` call and so on. We have to merge the resulting call two more times until the “temporary increase” has gone and the termination proof has become trivial.

7 Conclusion

We have described how dependency pair proofs can be performed in a relational setting. Our approach is more appropriate for termination proofs about functions that are defined by wellfounded recursion in an interactive theorem prover.

The transformations described here are easy to automate in the form of tactics, and we have already implemented some of them. Of course, finding suitable orderings for the simplified problems is still difficult, but heuristics based on measure functions can be applied. In fact, our approach naturally subsumes the
existing automation based on lexicographic combinations [3], which can be seen as a repeated application of reduction pairs, but without using the dependency graph.

Perhaps more interesting is that we gave a very simple and non-technical interpretation of (some aspects of) the dependency pair method outside the framework of term rewriting, but in terms of general lemmas about wellfoundedness, union and composition. Compared to term rewriting, the notions in our framework are often slightly more abstract and, in a sense, less operational: For instance, the $\rightarrow^*_R$ relation in the definition of a chain has been replaced by just equality, and the requirement that reduction pairs should be monotonic [8] is dropped altogether. A more detailed analysis of the differences between our approach and traditional dependency pairs is the subject of future work.

Acknowledgement I thank Pierre Courtieu, Cathrine Dubois, and Xavier Urbain for inviting me to Paris and for interesting discussions that stimulated the ideas presented here. Amine Chaieb and Christian Sternagel gave valuable feedback on a draft of this paper.

References

A Reflexive Formalization of a SAT Solver in Coq

Stéphane Lescuyer\textsuperscript{1,2} and Sylvain Conchon\textsuperscript{2,1}

\textsuperscript{1} INRIA Saclay-Île de France, ProVal, Orsay F-91893
\textsuperscript{2} LRI, Université Paris-Sud, CNRS, Orsay F-91405

Abstract. We present a Coq formalization of an algorithm deciding the satisfiability of propositional formulas (SAT). This SAT solver is described as a set of inference rules in a manner that is independent of the actual representation of propositional variables and formulas. We prove soundness and completeness for this system, and instantiate our solver directly on the propositional fragment of Coq’s logic in order to obtain a fully reflexive tactic. Such a tactic represents a first and important step towards our ultimate goal of embedding an automated theorem prover inside the Coq system. We also extract a certified OCaml implementation of the algorithm.

1 Introduction

The fact that safety-critical software keeps getting bigger and more numerous has certainly contributed to a lot of effort being spent on certification systems in the recent years. However, improvements of automation techniques have only partially benefited to interactive provers in general, and to the Coq proof assistant \cite{coq} in particular. Although specific decision procedures have been and are still being implemented in Coq, they still lack co-operation with each other. For instance, tactics like \texttt{omega}, \texttt{tauto} and \texttt{congruence} respectively address linear arithmetic, propositional logic and congruence closure, but their combination still has to be driven manually by users, whereas tools like SMT solvers can perform this combination automatically.

We are currently developing an SMT solver dedicated to program verification \cite{SMT} and our goal is to embed it as a decision procedure in Coq. Not only would it validate the algorithms and schemes at work in our solver, but it would also provide the automatic interaction of dedicated decision procedures in Coq. In this paper, we present our first step towards this goal: the formalization of a SAT solver, the decision procedure that lies at the heart of our prover. We also use reflection to obtain a tactic that relies on this certified decision procedure.

In Section 2, we quickly recall what the DPLL procedure is and we formalize it by a set of inference rules. In Section 3, we describe a Coq formalization of this procedure and we prove its soundness and completeness. We then use this procedure in Section 4 in order to build a reflexive tactic solving propositional goals. We show examples of applications of this tactic and its limitations in Section 5, before concluding with a discussion of related work in Section 6.

2 The DPLL Procedure

The DPLL procedure \cite{dpll,dl}, named after its inventors Davis, Putnam, Logemann and Loveland, is one of the oldest decision procedure for the problem of checking the satisfiability of a propositional formula. DPLL deals with formulas in \textit{conjunctive normal form} (CNF), i.e. conjunction of clauses, where a clause is a disjunction of propositional literals. A formula in CNF can thus be written $\land_{i=1}^{n}(l_1 \lor \cdots \lor l_k)$ where each $l_j$ is a propositional variable or its negation.

DPLL essentially tries all possible valuations of the variables of a formula until it finds one that satisfies the formula. Since there is an exponential number of such valuations, DPLL enhances over the basic exploration of all valuations by the eager use of the following two simplifications:

- \textit{Boolean constraint propagation}: once a truth value has been chosen for a given variable, the literals that become false can be removed from their clauses, and if a clause contains a literal that becomes true, the whole clause can be removed from the formula since it is now known to be true;
Unit propagation: whenever a clause is reduced to a single literal, the valuation of this literal’s variable must be set so that the literal is true, otherwise the whole formula would be false; an efficient detection of such clauses can dramatically change the overall performance of industrial-strength SAT solvers [17].

When every possible simplification has been applied, the algorithm tries to assign a boolean value to a variable of its choice, which leads to more simplification. Eventually, the procedure reaches one of the following cases:

- the problem is empty, i.e. the formula has been reduced to the empty conjunction, in which case a valuation satisfying the original formula has been found and the algorithm terminates;
- the formula contains an empty clause, which means it is obviously unsatisfiable; in that case, DPLL backtracks to an earlier point, where it tries assigning another value to a variable.

![Fig. 1. The DPLL procedure seen as a proof derivation system](image-url)

We give a formalization of this DPLL procedure as a set of five inference rules, presented in Fig. 1. The current state of the algorithm is represented as a sequent $\Gamma \vdash \Delta$ where $\Gamma$ is the set of literals that are assumed to be true, and $\Delta$ is the current formula, seen as set of clauses, i.e. a set of sets of literals.

More precisely, we write $l \lor C$ for a clause that contains the literal $l$, and $l_1, l_2, l_3$ for the set of literals $\{l_1, l_2, l_3\}$. We keep the braces around unit clauses so as to avoid confusion, e.g. $\{l\}$ for the clause containing only $l$. On the right-hand side of the sequent, we denote by $\Delta, C$ the conjunction of a clause $C$ and a CNF formula $\Delta$. Finally, the negation of a literal $l$ is denoted $\overline{l}$, with $\overline{\overline{l}} = l$.

We now take a closer look at the rules of Fig. 1, which should be read bottom-up. The CONFLICT rule corresponds to the case where a clause has been reduced to the empty clause: this rule terminates a branch of the proof search and forces the algorithm to backtrack in order to find another valuation. UNIT implements unit propagation: if a clause is reduced to the literal $l$, this literal can be added to the context before proof search goes on. ELIM and RED each perform one kind of boolean constraint propagation: if the negation of a literal is in $\Gamma$, it can be removed from all clauses (RED); if a clause contains a literal that is supposed true, the whole clause can be removed (ELIM). The last rule is the rule that actually performs the branching, and thus the “proof search”. SPLIT picks any literal $l$ and adds it to the context $\Gamma$. If no instantiation is found on this side (i.e. all the branches end with CONFLICT), then $l$ is supposed true instead and the right branch is explored.

If there exists a derivation for a formula $F$ starting with an empty context $\emptyset \vdash F$, this means that the whole tree has been explored and no model has been found, and therefore that the formula $F$ is unsatisfiable.

3 Formalizing DPLL in Coq

In this section, we present a Coq formalization of the system presented in Section 2, for which we prove soundness and completeness with respect to a notion of semantics for formulas.

3.1 Preliminary Definitions

We start by defining how literals and formulas shall be represented. To do so, we will make use of Coq’s module system [5, 2]. Coq module types allow one to pack together types, functions and related axioms by
keeping a high level of abstraction. One can then create functors, i.e. modules which are parameterized by other modules of a certain signature and which can then be instantiated on any modules that match the expected signature.

---

Module Type LITERAL.
  Parameter t : Set.
  (* Stuff literals should provide *)
  Parameter mk_not : t → t.
  Axiom mk_not_invol : ∀ l, mk_not (mk_not l) = l.
  ...
  (* t is an ordered type *)
  Parameter eq : t → t → Prop.
  Parameter lt : t → t → Prop.
End LITERAL.

Fig. 2. A module type for literals

Therefore, by taking advantage of Coq’s module system, it is sufficient to define module types for literals and formulas, and we can then develop our decision procedure in a way that is independent of the actual representation of the input. For instance, the module type LITERAL (Fig. 2) provides a type t for literals, a function mk_not that builds the negation of a literal and some axioms about this function (like the fact that it is involutive). Literals also come with a decidable equality and a total order, which are necessary to later define finite sets of literals.

Once literals are defined, we can similarly define a module type CNF for formulas, as shown in Fig. 3. Such a module shall of course provide a type formula for formulas, and embed a module of type LITERAL. It also comes with two modules of finite sets\(^3\): LSet, whose elements are literals, and CSet, whose elements are sets of literals, as can be seen in the with Module E := ... part of their definitions. Finally, a module for formulas comes with a function make, that transforms a formula into a set of sets of literals; in other words, the function make performs conversion into conjunctive normal form.

---

Module Type CNF.
  Parameter formula : Set.
  Declare Module L : LITERAL.
  Declare Module LSet : FSetInterface.S with Module E := L.
  Parameter make : formula → CSet.t.
End CNF.

Fig. 3. A module type for formulas

We can now start the definition of a functor SAT parameterized by a module F of type CNF and which will implement our SAT solving algorithm without any knowledge about the actual formulas. The development can only use elements that are defined in F’s signature and this ensures modularity as well as reusability.

---

\(^3\) FSetInterface.S is an interface for finite sets from the standard Coq library; it contains a module E which is the module of its elements.
Fig. 4 shows the beginning of the module, as well as the definition of sequents: a sequent, noted \( G \vdash D \), is simply a record with a set of literals \( G \) and a set of clauses \( D \), as discussed in Section 2.

```
Module SAT (F : CNF).
  Import F.

  Record sequent : Set := {G : LSet.t; D : CSet.t}.
```

Fig. 4. The SAT module and a definition of sequents

The next step is the definition of the rules system presented in Fig. 1. We use an inductive definition shown in Fig. 5 by enumerating all possible ways a derivation can be built from a given sequent. We call this inductive derivable and an object of type derivable \((G \vdash D)\) represents a proof derivation of sequent \( G \vdash D \). Note that each constructor faithfully follows from a rule of the original system. For instance, AUnit describes unit propagation, and AElim and ARed together describe the two \( \text{BCP} \) rules.

```
Inductive derivable : sequent → Set :=
| AConflict : ∀ G D, ∅ ∈ D → derivable (G ⊢ D)
| AUnit : ∀ G D l, \{l\} ∈ D → derivable (G, l ⊢ D \ \{l\}) → derivable (G ⊢ D)
| AElim : ∀ G D l C, l ∈ G → l ∈ C → C ∈ D → derivable (G ⊢ D \ \{C\}) → derivable (G ⊢ D)
| ARed : ∀ G D l C, l ∈ G → \¬ l ∈ C → C ∈ D → derivable (G ⊢ D \ \{C\}, C \ \{l\}) → derivable (G ⊢ D)
| ASplit : ∀ G D l, derivable (G, l ⊢ D) → derivable (G, \¬ l ⊢ D) → derivable (G ⊢ D).
```

Fig. 5. The inductive definition of the proof system

3.2 Semantics

In the previous subsection, we defined literals, formulas and what it means for a sequent to be derivable. To show the correctness of this derivation system, we need a notion of semantics, i.e. what it means for a formula to be “true”. We cannot directly (nor do we want to) rely on the prover’s notion of truth because we are dealing with abstract formulas and not native Coq propositional formulas.

Once again we use Coq’s functorization system and define semantics as a functor with respect to a module \( F \) of type \( \text{CNF} \). A model is simply defined as a function assigning a propositional value \( \text{Prop} \) to any literal:

```
Definition model := \{M : L.t → Prop | ∀ l, M l ↔ \¬ (M \¬ l)\}.
```

We use a dependent type to ensure that models are only functions which have the reasonable property of not assigning the same truth value to a literal and its negated counterpart. We can then use (via a coercion) a model \( M \) as a function interpreting a literal to an element of the propositional sort \( \text{Prop} \). It is straightforward to define what it means for a model to satisfy a clause or a set of clauses, and when a formula is unsatisfiable:

\footnote{In this figure and in the following, we use mathematical notations for set-related operations, rather than Coq’s concrete syntax, for the sake of readability.}
Definition sat_clause (M : model) (C : LSet.t) := ∃ l ∈ C, M l.
Definition sat_goal (M : model) (D : CSet.t) := ∀ C ∈ D, sat_clause M C.
Definition unsatisfiable (D : CSet.t) := ∀ (M : model), ~ sat_goal M D.

This gives us a notion of satisfiability for clauses and formulas, but we also need to take the context of a sequent into account. What does it mean for a sequent G ⊢ D to be “unsatisfiable”? The context G of such a sequent represents a partial assignment and instead of considering any model when checking the satisfiability of D, it means that we have to consider only models that entail G. This leads to a notion of “submodel” that we define as follows:

Definition submodel (G : LSet.t) (M : model) := ∀ l ∈ G, M l

Note that this definition of a submodel implies that G is a valid partial assignment, in the sense that it does not contain both a literal and its negation. From this notion of submodel naturally follows the correct definition of unsatisfiability for a sequent, which we call incompatibility, and states that there is no model of the context that also satisfies the clauses on the right-hand side:

Definition incompatible (G ⊢ D : sequent) := ∀ (M : model), submodel G M → ~ sat_goal M D.

3.3 The Decision Procedure

Using the semantics we just defined, we can now proceed to prove the fundamental theorems about our derivation system. First in line is the soundness of the proof system:

if there exists a derivation of the sequent ∅ ⊢ D, D is unsatisfiable

We actually prove something more general than this statement, using the notion of incompatibility that we just described:

Theorem soundness : ∀ S : sequent, derivable S → incompatible S.

The special case where the context of sequent S is empty yields exactly the above statement. This theorem can be proved by a structural induction on the derivation of S: for each case, it is sufficient to show that if the premises are incompatible, then so is S. We gave the informal arguments when describing the DPLL procedure in Section 2. The Coq proof is not difficult (about 50 lines of tactics).

Conversely, the completeness of the algorithm could be expressed by the following statement:

Theorem completeness : ∀ S : sequent, incompatible S → derivable S.

There are at least two reasons why we do not prove completeness in this particular form:

− We do not only want full equivalence between the notions of derivability and incompatibility, but we also want a decision procedure, i.e. a function capable of telling if a given formula is unsatisfiable or not. Proving such a theorem of completeness would certainly give us an equivalence between the derivability of a sequent and its incompatibility, thus bringing the problem of deciding satisfiability down to the one of deciding derivability. However, deciding derivability amounts to try and build a derivation for a given sequent if possible, and it is a proof that actually encompasses the completeness theorem presented above. Thus, we want to avoid doing the same job twice.

− Not only do we need a decision procedure that we could extract, but we want to be able to use that procedure in Coq through the mechanism of reflection, i.e. by actually computing the proof search in the system. It is well known that procedures with propositional contents cannot be executed as efficiently as computational-only functions, because in the first case, proofs need to be replayed along with computations. Thus, we do not want to encode the decision procedure as part of a general completeness theorem.
For these reasons, we will build the decision procedure in two steps: first we will program a function without propositional content to implement the actual decision procedure, and then we will show that its results are correct. This function will not return any “complex” information, but only \textit{Sat} \(G\) if it has found a partial model \(G\), and \textit{Unsat} otherwise:

\[
\begin{align*}
\text{Inductive } \text{Res} & : \text{Set} := \\
\text{Sat} & : \text{LSet.t} \to \text{Res} \\
| \text{Unsat}.
\end{align*}
\]

The decision procedure \textit{per se} can now be implemented as a recursive function returning such a result:

\[
\begin{align*}
\text{Fixpoint } \text{proof_search} \ (G \vdash D : \text{sequent}) \ n \ \{\text{struct } n\} : \text{Res} := \\
\text{match } n \text{ with} \\
| 0 \Rightarrow \text{Sat } \emptyset \text{ (Absurd case)} \\
| S \ n_{D} \Rightarrow \\
\text{ if } D = \emptyset \text{ then Sat } G \text{ (* Model found! *)} \\
\text{ else } \\
\text{ if } \emptyset \in D \text{ then Unsat (* Rule AConflict *)} \\
\text{ else } ...
\end{align*}
\]

Because the recursion is not structural, we use an extra integer argument \(n\), and we will later make sure that we call the function with an integer large enough so that \(n\) never reaches 0 before the proof search is completed. This short excerpt of the function \textit{proof_search} shows that it proceeds by trying to apply some rules one after another, with a given \textit{strategy}. Here, the function first checks if the problem is empty, in which case it returns the current context as a model; otherwise, it checks if the empty clause is in the formula, in which case it returns \textit{Unsat}. We do not explicit the strategy further because it has no real interest in the scope of this paper.

The first theorem about \textit{proof_search} states that when it returns \textit{Unsat}, it indeed constructed a derivation on the way:

\[
\begin{align*}
\text{Theorem } \text{proof_unsat} : \forall \ S, \text{proof_search } S \ n = \text{Unsat} \rightarrow \text{derivable } S.
\end{align*}
\]

The proof follows the flow of the function and shows that each recursive call that was made corresponds to a correct application of the derivation rules. One may wonder why we didn’t construct this derivation in \textit{proof_search}, so as to return it with \textit{Unsat}: the reason is that a derivation contains proofs (in side conditions) and had we done so, our function would not have been 100% computational anymore.

The second theorem about \textit{proof_search} is the one that encompasses completeness: it states that if \textit{Sat} \(M\) has been returned, it is indeed a model of the formula and of the context.

\[
\begin{align*}
\text{Theorem } \text{proof_sat} : \forall n \ S \ M, \ \mu(S) < n \rightarrow \text{wf_context } S.G \rightarrow \text{proof_search } S \ n = \text{Sat } M \rightarrow \\
S.G \subseteq M \land \text{sat_goal } M \ S.D.
\end{align*}
\]

A couple of remarks about this theorem are necessary:

- \(\mu\) is a \textit{measure} of a sequent that we have defined in Coq, and for which we proved that it decreases for every recursive call in the algorithm. We could have defined the function by a well-founded induction on this measure, but it is computationally slightly more efficient to use the extra integer. This is a well-known technique to transform non-structural inductions in structural inductions [1]. When calling \textit{proof_search} on a sequent \(S\), a suitable integer is \(\mu(S) + 1\);

\footnote{Technically, the set returned is not a model because it is only partial; it can be completed into a model though, as long as it is a valid partial assignment, and we simplified the actual details here since they seem cumbersome.}

69
we need an extra hypothesis that the context remains well-formed (\texttt{wf context S.G}), which means that it doesn’t contain a literal and its negation. This is not guaranteed by the derivation rules because the side conditions were purposely very loose in order to allow any kind of strategy. Here, it is our strategy that guarantees this invariant is never broken, and this is part of the completeness proof.

Together with the soundness theorem, these two theorems show that \texttt{proof_search} is a decision procedure for unsatisfiability and we can now prove the following theorem:

\textbf{Theorem} \texttt{dpll_dec (Δ : CSet.t)}:

\{\texttt{incompatible (∅ ⊢ Δ)}\} + \\{\texttt{~incompatible (∅ ⊢ Δ)}\}.

The definition of \texttt{proof_search} and the proofs of its properties require 700 lines of code.

4 Deriving a Reflexive Tactic

The procedure we have developed so far can be extracted to an OCaml functor (cf. Section 5), but we are also greatly interested in directly using this procedure as a tactic to solve goals in our proof assistant.

4.1 Reification

In order to use our SAT solver on Coq propositional formulas, we need to instantiate the \texttt{SAT} functor. This raises the question of the actual representation of formulas and literals: we need to build modules of types \texttt{LITERAL} and \texttt{CNF} that will represent Coq formulas.

A natural choice for the type of literals would be to directly use the type \texttt{Prop} of propositions, but this is impossible because we need to build sets of literals, and more generally we need to be able to decide if two given propositions are equal or not. Indeed, consider the formula $\texttt{A} \land \texttt{~A}$; we need to know that the propositional variable $\texttt{A}$ is the same on both sides to conclude that this formula is unsatisfiable. Since the only decidable equality on sort \texttt{Prop} is the one that is always true, we cannot use \texttt{Prop} as the type of literals.

Instead, we resort to Coq’s metalanguage Ltac [8]. This language provides pattern-matching facility on Coq terms, and thereby allows us to check the syntactic equality of propositional terms at a metalevel. We will use this language to build, for a given propositional formula $F$, an \textit{abstract representation} of $F$ on which we will be able to apply the algorithm. This process, called \textit{reification} or sometimes \textit{metaification}, has already been used and described in [9, 11].

Using Ltac, we first build a function \texttt{get_vars} which traverses a formula $F$ and retrieves a list of all the propositional variables of $F$. We define another function \texttt{list_to_map} that turns such a list into a balanced map. This map now contains all the propositional variables of $F$ and provides an efficient way to search for a particular variable into a map. For instance, if $F$ is the following formula:

\[ F: \texttt{A} \land (\texttt{~B} \lor (p \texttt{A C}) \land (\forall \texttt{D}, (p D D))). \]

the result of \texttt{list_to_map (get_vars F)} will be a map containing the variables $\texttt{A}$, $\texttt{B}$, $\langle \texttt{p A C} \rangle$ and $\forall \texttt{D}$, $\langle \texttt{p D D} \rangle$. In particular, the last variable is abstracted because our propositional language does not include quantifiers. Given this map, we are able to represent variables by their \textit{path} in the map. It is now straightforward to create the module \texttt{LPROP} of literals, where a literal is just a \textit{path} in the map and a boolean saying if it is negated or not, and the \texttt{mk_not} function a simple inversion of this boolean:

\begin{verbatim}
Module LPROP <:< LITERAL.
  Inductive path : Set := Lft : path → path | Rgt : path → path | E.
  Definition t := path × bool.
  Definition mk_not (p,b) : t := (p, negb b).
End LPROP.
\end{verbatim}

70
We can move on to defining the corresponding types for formulas. We will for now assume that our formulas are already in conjunctive normal form, and we address the problem of conversion to CNF later in Section 4.2. In Fig. 6, we show an excerpt of the module CNFPROP of type CNF, which implements our type of formulas. Its literals are, of course, the literals of the module LPROP we just defined. Formulas and clauses are defined in a very natural way by two inductives: a formula is either a clause or a conjunction of formulas; a clause is a literal or a disjunction of clauses. This representation makes the function make converting a formula to a set of sets of literals (not represented here) really straightforward.

```coq
Module CNFPROP <: CNF.
Module L := LPROP.

Inductive clause : Set :=
  | COr : clause → clause → clause
  | CLit : L.t → clause.

Inductive formula : Set :=
  | FAnd : formula → formula → formula
  | FClause : clause → formula.
...
End CNFPROP.
```

Fig. 6. A module for propositional formulas

We also define an interpretation function interp such that interp v f interprets an object f of type formula to its propositional counterpart in Coq. The extra argument v is the map binding paths to concrete propositional variables, which is needed to interpret literals. In particular, interp uses the following subroutine to interpret literals, where lookup id v returns the proposition bound to id in the map v:

```coq
Definition linterp (l : L.t) : Prop :=
  match l with
  | (id, true) ⇒ lookup id v
  | (id, false) ⇒ ~(lookup id v) end.
```

The last step of the reification process is to build a tactic in Ltac, that, for a given formula F in Coq’s propositional language, builds an abstract formula f of type formula and a map v such that interp v f = F. We have already covered the construction of the map v. The construction of the formula f is realized by a couple of recursive Ltac tactics which analyze the head symbol of the current formula to construct the corresponding abstract version. For instance, the top-level function matches conjuncts and goes like this:

```coq
Ltac reify_formula F v :=
  match constr:F with
  | and ?F1 ?F2 ⇒
    let f1 := reify_formula F1 v with f2 := reify_formula F2 v in
    constr:(FAnd f1 f2)
  | ?F ⇒
    let c := reify_clause F v in constr:(FClause c)
  end.
```

Now, if we go back to our previous example, and if we take this formula as our current goal, we can use the tactics we just described to build a suitable map, reify the goal in an abstract formula f, and replace the current goal by the interpretation of f. The tactic change asks Coq to perform the so-called conversion rule: it computes the interpretation and checks that it is indeed equal to the original goal.

```
A ∧ (~ B ∨ (p A C)) ∧ (∀D : Prop, (p D D))
```

71
match goal with | ⊢ ?F ⇒
  let v := list_to_map (get_vars F) in
  let f := reify_formula F v in
  change (interp v f)
end.

===============
interp (...) (FAnd (FClause ...) (FAnd ... ...))

4.2 The Generic Tactic

At this point, in order to turn our development into a user-friendly generic tactic, we still need to address a couple of issues.

Conversion to normal form. Before running the actual proof search, a formula should be put in CNF. If it is not in CNF, then some subformulas will be abstracted (like the quantified part in our example above). We could have coded the conversion to normal form as a Coq function running on reified formulas, but we decided to avoid this additional tedious work by again using tacticals. Coq provides a tactic named autorewrite which performs automatic rewriting of expressions. When fed with a set of (oriented) equalities describing a normalizing system, autorewrite will transform an expression into its normal form with respect to this system. Thus, we encode the conversion into CNF as a set of rewriting rules\(^6\): linearizing implications, pushing negations to the atomic variables, distributing disjunction over conjunction, etc. Some of these rules only hold in classical logic, and we discuss this further in Section 5.

Lifting the Semantics. In Section 3.2, we defined a notion of semantics and proved the properties of our decision procedure in this respect. Recall that a model is a function associating a propositional variable to every literal. Thus, it turns out that our abstract formulas have a very canonical notion of model: the interpretation of the literals itself. Indeed, if \(l\) is a literal representing a variable \(A\) of type Prop, the canonical model satisfies \(l\) if and only if there is a proof of \(A\). This result lifts to clauses and formulas, and we can prove this adequation:

**Theorem adequation :**

\[ \forall (f : \text{formula}), \ \text{interp} \ v \ f \rightarrow \text{sat_goal} \ (\text{model} \ v) \ (\text{make} \ f). \]

where model v is the canonical model interpreting literals in the map v. This theorem can be read as: “if there is a proof of a formula \(F\), then its reified counterpart \(f\) is satisfiable”. Together with the soundness of the decision procedure, this gives us the following fact:

**Corollary validity :**

\[ \forall (f : \text{formula}), \ \text{proof_search} \ (\emptyset \vdash \ (\text{make} \ f)) = \text{Unsat} \rightarrow ^{\sim} \text{(interp} \ v \ f). \]

We can now wrap everything up in a high-level tactic unsat that builds the conjunction \(F\) of all the hypotheses in the context, turns it into normal form, constructs the abstract version \(f\) of \(F\), changes \(F\) in the context to interp \(v \ f\), and finally applies the validity theorem to bring the current goal down to a proof of proof_search \( (\emptyset \vdash (\text{make} \ f)) = \text{Unsat} \). Coq is then asked to compute the left-hand side of this equation, which triggers the actual proof search. If the procedure returns Unsat, the goal is trivial and the proof is completed. Otherwise, it returns a countermodel and we print it out, since it can be very valuable to the user in order to understand why the tactic did not succeed.

The same mechanism can also be used to prove the validity of a current goal \(F\), by applying double negation and trying the unsat tactic on \(^{\sim} F\). We provide a tactic called valid that performs these operations. The definitions and proofs for unsat and valid represent about 400 lines.

\(^6\) In practice, we use several complementary rewriting systems, because for efficiency reasons, some transformations must be done before others, e.g. rewriting of implications.
5 Results and Examples

We start this section by giving a small example of how the tactic \texttt{unsat} can be used in practice. Suppose our goal is the following propositional formula where variables \(A\) to \(D\) have type \texttt{Prop}:

\[
\begin{align*}
\begin{array}{c}
\text{===>} \\
\text{unsat.}
\end{array}
\end{align*}
\]

If we try to apply \texttt{unsat} to this goal, the tactic will try to show that the left-hand side of the implication is unsatisfiable. Since it is not, the tactic fails and prints out the countermodel shown above: indeed, one can easily verify that this valuation makes the goal false. We can use this countermodel to add complementary hypotheses to our formula, for instance that \(B\) is true and \(A\) is false. By doing so, we see that the \texttt{unsat} tactic now succeeds in about one tenth of a second:

\[
\begin{align*}
\begin{array}{c}
\text{===>} \\
\text{Time unsat.}
\end{array}
\end{align*}
\]

Our experiments with this tactic show that goals that occur in practice during an interactive proof are reasonably small and the biggest part of the time needed to prove a goal is often spent in the conversion into CNF. We also tried our tactic on artificial benchmarks from the SATLIB initiative [14], in comparison to the performance of the extracted SAT solver, and the results are summarized in Fig. 7. The second row gives the number of variables and clauses in each problem. The fourth row gives the time spent by the tactic on the reification alone. The last two rows respectively show the number of nodes in the proof derivation, and the number of \texttt{Split} nodes. These results show that both the tactic and the extracted implementation perform rather badly on these tests. In some cases, the tactic was unable to succeed in reasonable time (less than a couple of hours), even though the input problems were already in CNF. This can be explained by the fact that we used a very unoptimized version of the DPLL procedure and that these benchmarks are especially good at stressing optimizations of DPLL. The high number of branching nodes illustrates the fact that we are exploring a great part of the search tree.

An important characteristic of the DPLL procedure is that its performance heavily depends on how decision literals, i.e. literals that are introduced in the rule \texttt{Split}, are chosen. Considering this, we took advantage of the modularity of our development to allow the user to instantiate our SAT solver with a customized function called \texttt{pick}, and coming with the module \texttt{CNF}. This function \texttt{pick} is used by the

<table>
<thead>
<tr>
<th></th>
<th>aim-1</th>
<th>aim-2</th>
<th>aim-3</th>
<th>aim-4</th>
<th>phole 5</th>
<th>phole 6</th>
<th>phole 7</th>
<th>phole 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>vars/clauses</td>
<td>50/80</td>
<td>50/80</td>
<td>50/80</td>
<td>50/80</td>
<td>30/31</td>
<td>42/133</td>
<td>56/204</td>
<td>72/297</td>
</tr>
<tr>
<td>OCaml time</td>
<td>7m20s</td>
<td>33s</td>
<td>7m10s</td>
<td>1m50s</td>
<td>0.3s</td>
<td>2.0s</td>
<td>25s</td>
<td>6m</td>
</tr>
<tr>
<td>Coq time</td>
<td>-</td>
<td>2h30m</td>
<td>-</td>
<td>-</td>
<td>15s</td>
<td>3m20s</td>
<td>1h9m</td>
<td>-</td>
</tr>
<tr>
<td>whereof reification</td>
<td>25s</td>
<td>22s</td>
<td>26s</td>
<td>25s</td>
<td>3.5s</td>
<td>10s</td>
<td>24s</td>
<td>56s</td>
</tr>
<tr>
<td>Nodes</td>
<td>13M</td>
<td>1.3M</td>
<td>33M</td>
<td>6.8M</td>
<td>12k</td>
<td>111k</td>
<td>1.2M</td>
<td>15M</td>
</tr>
<tr>
<td>Branching</td>
<td>630k</td>
<td>64k</td>
<td>2M</td>
<td>400k</td>
<td>370</td>
<td>3250</td>
<td>32k</td>
<td>378k</td>
</tr>
</tbody>
</table>

Fig. 7. Some results on harder SAT problems
procedure to pick a literal from the current goal when applying the rule Unsat. The correctness of the procedure only relies on a couple of easy properties that the user shall prove for his customized function pick. This way, the SAT solver does not depend on particular heuristics and new efficient heuristics can be implemented by the front-end user.

Another advantage of our modular approach comes to light when considering whether our procedure works in classical or intuitionistic logic. A SAT solver for propositional logic is inherently classical, as the rule Split, for instance, relies on the fact that a literal must be either true or false. Nonetheless, some instances of the excluded-middle are provable in intuitionistic logic and there may be cases where the decision literals used in the proof search are actually decidable (e.g. membership of an element in a finite set). For this reason, we made sure that our SAT functor did not depend on the excluded-middle, but that we only put the restriction on models instead: a model $M$ shall have the property that, for any literal $l$, $M\ l \lor \neg M\ l$ is provable. Of course, when creating the tactics unsat and valid, we were dealing with any kind of propositional atoms, and therefore we had to assume the excluded-middle. Another possible approach, but that we have not implemented yet, would be to generate one subgoal for every decision literal in the proof and let the user deal with them. To ensure that users proving goals with our generic tactics are aware of the fact that they are working in classical logic, we encapsulated the top-level tactic definitions in a functor LoadTactic expecting the classical axiom in input:

```coq
Module Type K.
  Axiom classic : \forall P, P \lor \neg P.
End K.
Module LoadTactic (K : K).
  Import K.
  ...
End LoadTactic.
```

To further illustrate our point, it is possible to instantiate our procedure on boolean formulas: the difference between propositional and boolean formulas in Coq is that booleans are a type in $\text{Set}$, whose equality is decidable and which can be extracted to ML booleans. In particular, a canonical model for booleans would be:

```coq
Definition model (b : bool) := if b then True else False.
```

and such a model has the property required by the SAT solver. Therefore, it is possible to use our SAT solver to create a tactic deciding satisfiability of boolean formulas in intuitionistic logic by just following the same steps as in Section 4. The whole development presented so far, including these extra tactics on booleans, is available online\(^7\) and can be compiled with Coq v8.1pl3.

### 6 Related Work and Conclusion

**Related Work.** Decision procedures for classical propositional logic have already been formalized in proof assistants before. For instance, Harrison [13] on one side, and Letouzey and Théry [15] on the other side, both presented formalizations of Stålmarck’s algorithm, respectively in HOL [10] and Coq. In both cases, they extracted certified versions of the algorithm which also produced traces. The traces were then used to reconstruct a proof from inside the prover, yielding a “partially reflexive” tactic. In our work, we focused on developing a totally reflexive tactic in order to avoid the cost of reconstructing a proof from traces. Harrison showed how to implement Binary Decision Diagrams (BDDs) as a HOL derived rule [12], with pretty much the same approach as in [13]. Closer to our work, Verma et al. [16] implemented reflexive BDDs in Coq as a first step towards a reflexive model checker in Coq. Our approach is similar in the sense that our SAT solver is the cornerstone of the SMT solver we are planning to formalize.

\(^7\) [http://www.lri.fr/~lescuyer/sat/unsat.tgz](http://www.lri.fr/~lescuyer/sat/unsat.tgz)
Conclusion. We have presented the formalization in Coq of a DPLL-like procedure deciding satisfiability of propositional formulas. We extracted a certified OCAML implementation of this SAT solver, which is about 1700 lines long (including 800 lines for the interface). We also instantiated our procedure on Coq’s propositional formulas in order to derive a reflexive tactic. Even if its performance remains limited, our tactic can be used in Coq to automatically discharge valid goals or unsatisfiable contexts.

The use of Ltac allowed us to avoid coding parts of the tactic in OCAML, and we hope our development illustrates what we think is a key feature of Coq: the ability to reason and program about terms directly at the top-level, without any knowledge of its inner mechanisms, and without having to compile stubs coded in OCAML. We also showed how modularity can be beneficial, just as in a usual programming language. By using Coq’s module system to develop our procedure in a very modular way, we were able to use this procedure on different types of propositional formulas without much pain and to give the user the ability to define his own heuristics. Modularity also ensures that the decision procedure can be used in an intuitionistic setting on suitable data structures.

We plan to work in several directions, mainly by adding common optimizations to the DPLL procedure in order to prune some parts of the proof search. We are also greatly interested in combining this decision procedure with other, more specific, decision procedures like linear arithmetic or congruence closure.

References

Abstract. A Forensic Lucid intensional programming language has been proposed for intensional cyberforensic analysis. In large part, the language is based on various predecessor and codecessor Lucid dialects bound by the higher-order intensional logic (HOIL) that is behind them. This work formally specifies the operational aspects of the Forensic Lucid language and compiles a theory of its constructs using Isabelle, a proof assistant system.

1 Introduction

As a part of the Intensional Cyberforensics project, we define a functional-intensional programming/specification language, called Forensic Lucid. The language is under active design and development including its syntax, semantics, the corresponding compiler, run-time, and interactive “development” environments [1,2] that we refer to as General Intensional Programming System (GIPSY) [3]. We approach the problem using Isabelle [4] as a proof assistant system.

Problem Statement. A lot of intensional dialects have been spawned from the functional intensional programming language called Lucid [5,6,7,8,9,10,11,12]. Lucid (see Section 1.2) itself was invented with a goal for program correctness verification [7,8]. While there were a number of operational semantics rules for compiler and run-time environments developed for all those dialects throughout the years, there was no a complete formal proof set of the rules of the languages. Yet another dialect of Lucid has been created to foster the research on intensional cyberforensics (see Section 1.3), called Forensic Lucid, which, in a large part is a union of the syntax and operational semantics rules from the comprising languages with the forensic extensions. In order to be a credible tool to use, for example, in court, to implement relevant tools for the argumentation, the language ought to have a solid scientific base, a part of which is formalizing the semantics the language and proving correctness of the programs written in it.

Proposed Solution. In this work, we propose to begin validation of the Forensic Lucid constructs with the Isabelle prover assistant [4] and extend it to the comprising Lucid dialects as a whole. We proceed bottom-up from “core” Lucid dialects such as GIPL, Lucx, and Indexical Lucid and even their smaller decompositions as well as top-down from Forensic Lucid to arrive to a comprehensive set of proofs covering the dialects.

1.1 Intensional Logics and Programming

Definitions. Intensional programming (IP) is based on intensional (or multidimensional) logics, which, in turn, are based on natural language understanding aspects (such as time, belief, situation, and direction). IP brings in dimensions and context to programs (e.g. space and time in physics or chemistry). Intensional logic adds dimensions to logical expressions; thus, a non-intensional logic can be seen as a constant or a snapshot in all possible dimensions. Intensions are dimensions at which a certain statement is true or false (or has some other than a Boolean value). Intensional operators are operators that allow us to navigate within these dimensions. Higher-order intensional logic (HOIL) is the one that couples functional programming as that of Lucid with multidimensional dataflows that the intensional programs can query an alter through an explicitly notion of contexts as first-class values [13,14].
An Example of Using Temporal Intensional Logic. Temporal intensional logic is an extension of temporal logic that allows to specify the time in the future or in the past.

1. Let's take $E_1$ from (1) above. Then let us fix here to Montreal and assume it is a constant. In the month of February, 2008, with granularity of day, for every day, we can evaluate $E_1$ to either true or false:

   Tags: 1 2 3 4 5 6 7 8 9 ...

   Values: F F T T T F F F T ...

   If one starts varying the here dimension (which could even be broken down to X, Y, Z), one gets a two-dimensional evaluation of $E_1$:

   City: / 1 2 3 4 5 6 7 8 9 ...

   Montreal F F T T T F F T ...

1.2 Lucid

Lucid [5,6,9,7,8] is a dataflow intensional and functional programming language. In fact, it is a family of languages that are built upon intensional logic (which in turn can be understood as a multidimensional generalization of temporal logic) involving context and demand-driven parallel computation model. A program written in some Lucid dialect is an expression that may have subexpressions that need to be evaluated at certain context. Given the set of dimension $D = \{dim_i\}$ in which an expression varies, and a corresponding set of indexes or tags defined as placeholders over each dimension, the context is represented as a set of $<dim_i:tag_i>$ mappings and each variable in Lucid, called often a stream, is evaluated in that defined context that may also evolve using context operators [14,15,16,13]. The generic version of Lucid, GIPL [11], defines two basic operators $\otimes$ and $\#$ to navigate in the contexts (switch and query). The GIPL was the first generic programming language of all intensional languages, defined by the means of only two intensional operators $\otimes$ and $\#$. It has been proven that other intensional programming languages of the Lucid family can be translated into the GIPL [11]. Please refer to Appendix A for the greater details about Lucid origins, variables as streams, random access to streams, and the basic operators. Since the Lucid family of language thrived around intensional logic that makes the notion of context explicit and central, and recently, a first class value [16,13,14,15] that can be passed around as function parameters or as return values and have a set of operators defined upon. We greatly draw on this notion by formalizing our evidence and the stories as a contextual specification of the incident to be tested for consistency against the incident model specification. In our specification model we require more than just atomic context values – we need a higher-order context hierarchy to specify different level of detail of the incident and being able to navigate into the “depth” of such a context. A similar provision by has already been made by the author [17] and earlier works of Swoboda et al. in [18,19,20,21] that needs some modifications to the expressions of the cyberforensic context.

Some other languages can be referred to as intensional even though they may not refer to themselves as such, and were born after Lucid (Lucid began in 1974). Examples include hardware-description languages (HDLs, appeared in 1977) where the notion of time (often the only “dimension”, and usually progresses only forward), e.g. Verilog and VHDL. Another branch of newer languages for the becoming popular is aspect-oriented programming (AOP) languages, that can have a notion of context explicitly, but primarily focused on software engineering aspect of software evolution and maintainability.

1.3 Cyberforensic Analysis

Cyberforensic analysis has to do with automated or semi-automated processing of and reasoning about electronic evidence, witnesses, and other details from cybercrime incidents (involving computers, but not limited to them). Analysis is one of the phases in cybercrime investigation, where the others focus on evidence collection, preservation, chain of custody, information extraction that precede the analysis. The phases the follow the analysis are formulation of a
report and potential prosecution, typically involving expert witnesses. There are quite a few techniques, tools (hard-
ware and software), and methodologies have been developed for all the briefly mentioned phases of the cybercrime
investigation. A lot of attention has been paid to the tool development for evidence collection and preservation; a few
tools have been developed to aid “browsing” data in the confiscated storage media, log files, memory, and so on. A lot
less number of tools have been developed for case analysis of the data, and the existing commercial packages (e.g. En-
case or FTK) are very expensive. Even less so there are case management, event modeling, and event reconstruction,
especially with solid formal theoretical base. The first formal approach to the cybercrime investigation was the finite-
state automata (FSA) approach by Gladyshev et. al [22,23]. The approach is complex to use and understand for non
computer science or equivalent investigators. The aim of Forensic Lucid is to alleviate those difficulties, be sound and
complete, expressive and usable, and provide even further usability improvement with the graphic interface that allow
data-flow graph-based (DFG) programming that allows translation between DFGs and Lucid code for compilation and
is implemented for Indexical Lucid in GIPSY already [24], and requires forensic extensions. While Forensic Lucid is
in the design and implementation, its solid base is being established in part with this work. The goal of Forensic Lucid
in the cyberforensic analysis is to be able to express in a program form the encoding of the evidence, witness stories,
and evidential statements, that can be tested against claims to see if there is a possible sequence or multiple sequences
of events that explain a given story. This is designed to aid investigator to avoid ad-hoc conclusions and have them
look at the possible explanations the Forensic Lucid program execution would yield and refine the investigation, as
was shown in the works [22,23] investigators failed to analyze all the stories and their plausibility before drawing
conclusions in the case. We do not recite the cases here due to the length limitations.

2  Forensic Lucid

The end goal is to define our Forensic Lucid language where its constructs concisely express cyberforensic evidence,
which can be initial state of a case towards what we have actually observed as a final state. The implementing system
(i.e. GIPSY) has to backtrace intermediate results in order to provide the corresponding event reconstruction path, if
it exists. The result of the expression in its basic form is either true or false, i.e. “guilty” or “not guilty” given the
context per explanation with the backtrace. There can be multiple backtraces, that correspond to the explanation of the
evidence (or lack thereof).

2.1 Properties

We define Forensic Lucid to model the evidential statements and other expressions representing the evidence and
observations as a higher-order context hierarchy. An execution trace of a Forensic Lucid program would expose the
possibility of the proposed claim with the events in the middle.

Addition of the context calculus from Lucx for operators on Lucx’s context sets (union, intersection, etc.) are
used to address to provide a collection of traces. Forensic Lucid inherits the properties of Lucx, MARFL, Objective
Lucid, JOOIP (and their comprising dialects), where the former is for the context calculus, and the latter for the arrays
and structural representation of data for modeling the case data structures such as events, observations, and groupings
of the related data.

One of the basic requirements is that the complete definition of the operational semantics of Forensic Lucid should
be compatible with the basic Lucx and GIPL, i.e. the translation rules or equivalent are to be provided when imple-
menting the language compiler within GIPSY, and such that the GEE can execute it with minimal changes.

```plaintext
foo @
{
  [ final observed event, possible initial observed event ],
  [ ],
  [ ]
}
```

Listing 1.1. Intensional Storyboard Expression
While the [...] notation here may be confusing with respect to the notation of [dimension:tag] in Lucid and more specifically in Lucx [13,25], it is in fact a simple syntactical extension to allow higher-level groups of contexts where this syntactical sugar is later translated to the baseline context constructs. The tentative notation of {[...],...,[...]} implies a notion similar to the notion of the “context set” in [13,25] except with the syntactical sugar mentioned earlier where we allow syntactical grouping of properties, observations, observation sequences, and evidential statements as our context sets.

2.2 Transition Function

A transition function determines how the context of evaluation changes during computation. A general issue exists that we have to address is that the transition function $\psi$ is problem-specific. In the FSA approach, the transition function is the labeled graph itself. In the first prototype, we follow the graph to model our Forensic Lucid equivalent. In general, Lucid has already basic operators to navigate and switch from one context to another, which represent the basic transition functions in themselves (the intensional operators such as @, #, isood, first, next, fby, wvr, upon, and asa as well as their inverse operators$^1$). However, a specific problem being modeled requires more specific transition function than just plain intensional operators. In this case the transition function is a Forensic Lucid function where the matching state transition modeled through a sequence of intensional operators. In fact, the forensic operators are just pre-defined functions that rely on traditional and inverse Lucid operators as well as context switching operators that achieve something similar to the transitions in [22,23]. In fact, the intensional operators of Lucid represent the basic building blocks for $\psi$ and $\Psi^{-1}$.

2.3 Primitive Operators

The basic set of the classic intensional operators is extended with the similar operators, but inverted in one of their aspects: either negation of trueness or reverse of direction of navigation. Here we provide an informal definition followed by their formal counterpart of these operators alongside with the classical ones (to remind the reader what they do and enlighten the unaware reader). The reverse operators have a restriction that they must work on the bounded streams at the positive infinity. This is not a stringent limitation as the our contexts of observations and evidence in this work are always finite, so they all have the beginning and the end. What we need is an ability to go back in the stream and, perhaps, negate in it with classical-like operators, but reversed.

The operators are defined below to give a complete picture. The classical operators first, next, fby, wvr, upon, and asa were previously defined in [11] and earlier. The other complimentary, inverse, and negation operators were defined and revised from [26]. In this list of operators, especially the reverse ones, we make an important assumption that the streams we are working with are finite, which is sufficient for our tasks. Thus, our streams of context values can be bound between bod and eod and contain a finite tag set of elements is used as a context type. For summary of the application of the just defined operators’ examples, please refer to Appendix B.

Following the steps in [11], we further represent the definition of the operators via @ and #. Again, there is a mix of classical operators that were previously defined in [11], such as first, next, fby, wvr, upon, and asa as well as the new operators from this work. The collection of the translated operators denoted in monospaced font, while we provide their equivalence to the original Lucid operators, denoted as small caps.

The primitive operators are founding blocks to construct more complex case-specific functions that represent a particular investigation case as well as more complex so-called forensic operators.

- A stream of first elements of stream $X$:
  \[ \text{first} \; X = X@0 \]

- A stream of second elements of stream $X$:
  \[ \text{second} \; X = (x_1,x_1,...,x_1,...) = \text{first} \; \text{next} \; X \]

$^1$ Defined further.
- A stream of last elements of stream $X$:
  $\text{last } X = (x_n, x_n, ..., x_n, ...)$
  This definition of the $\text{last}$ operator relies on the earlier stated assumption that our streams can be explicitly finite for the language we are developing. This affects the follow up operators that rely on that fact just as well. It is also important to note that the $\text{last}$ operator in our design does not return $\text{eod}$ all the time on the finite stream due to lack of usefulness for such a value; instead it returns the element of the stream just before the $\text{eod}$.

\[
\text{last } X = X @ (#(\text{iseod} #) - 1)
\] (2)

- A stream of elements one before the last one of stream $X$:
  $\text{prelast } X = (x_n\text{□}_1, x_n\text{□}_1, ..., x_n\text{□}_1, ...)$
  $\text{prelast } X = \text{last } \text{prev } X$

- A stream of elements of stream $X$ other than the first:
  $\text{next } X = (x_1, x_2, ..., x_{i+1}, ...)$

\[
\text{next } X = X @ (# + 1)
\] (3)

- A stream of elements of stream $X$ other than the last:
  $\text{prev } X = (x_{n-1}, ..., x_{i+1}, x_i, x_{i-1}, ...)$

\[
\text{prev } X = X @ (# - 1)
\] (4)

- First element of $X$ followed by all of $Y$:
  $X \text{ fby } Y = (x_0, y_0, y_1, ..., y_{i-1}, ...)$

\[
X \text{ fby } Y = \begin{cases} X & \text{if } # = 0 \\ \text{else } Y @ (# - 1) & \end{cases}
\] (5)

- First element of $X$ preceded by all of $Y$:
  $X \text{ pby } Y = (y_0, y_1, ..., y_{i-1}, ..., y_n, x_0)$

\[
X \text{ pby } Y = \begin{cases} \text{if isbod } X & \text{then } X \text{ else } \text{prev } Y \\ \text{else } Y @ (# + 1) & \end{cases}
\] (6)

- Stream of negated arithmetic values of $X$:
  $\text{neg } X = (-x_0, -x_1, -x_2, ..., -x_{i+1}, ...)$

\[
\text{neg } X = -X
\] (7)

- Stream of inverted truth values of $X$:
  $\text{not } X = (!x_0, !x_1, !x_2, ..., !x_{i+1}, ...)$

\[
\text{not } X = \text{if } X \text{ then } X \text{ else } !X
\] (8)

- A logical AND stream of truth values of $X$ and $Y$:
  $X \text{ and } Y = (x_0 \&\& y_0, x_1 \&\& y_1, x_2 \&\& y_2, ..., x_{i+1} \&\& y_{i+1}, ...)$

\[
X \text{ and } Y = X \&\& Y
\] (9)

- A logical OR stream of truth values of $X$ and $Y$:
  $X \text{ or } Y = (x_0 \| y_0, x_1 \| y_1, x_2 \| y_2, ..., x_{i+1} \| y_{i+1}, ...)$

\[
X \text{ or } Y = X \| Y
\] (10)
A logical XOR stream of truth values of $X$ and $Y$:

$$X \text{ xor } Y = (x_0 \oplus y_0, x_1 \oplus y_1, x_2 \oplus y_2, \ldots, x_i \oplus y_i, \ldots)$$

$$X \text{ xor } Y = \text{not}((X \text{ and } Y) \text{ or } \text{not} (X \text{ or } Y)) \quad (11)$$

$\text{wvr}$ stands for *whenever*. $\text{wvr}$ chooses from its left-hand-side operand only values in the current dimension where the right-hand-side evaluates to $true$.

$$X \text{ wvr } Y =$$

$$\begin{align*}
& \text{if } \text{first } Y \neq 0 \\
& \text{then } X \text{ fby } (\text{next } X \text{ wvr next } Y) \\
& \text{else } (\text{next } X \text{ wvr next } Y)
\end{align*}$$

$$X \text{ wvr } Y = X @ T \text{ where}$$

$$\begin{align*}
T &= U \text{ fby } U @ (T + 1) \\
U &= \text{if } Y \text{ then } \# \text{ else next } U
\end{align*}$$

$\text{rwvr}$ stands for *retreat whenever*. $\text{rwvr}$ chooses from its left-hand-side operand backwards only values in the current dimension where the right-hand-side evaluates to $true$.

$$X \text{ rwvr } Y =$$

$$\begin{align*}
& \text{if } \text{last } Y \neq 0 \\
& \text{then } X \text{ pby } (\text{prev } X \text{ rwvr prev } Y) \\
& \text{else } (\text{prev } X \text{ rwvr prev } Y)
\end{align*}$$

$$X \text{ rwvr } Y = X @ T \text{ where}$$

$$\begin{align*}
T &= U \text{ pby } U @ (T - 1) \\
U &= \text{if } Y \text{ then } \# \text{ else prev } U
\end{align*}$$

$\text{nwvr}$ stands for *not whenever*. $\text{nwvr}$ chooses from its left-hand-side operand only values in the current dimension where the right-hand-side evaluates to $false$.

$$X \text{ nwvr } Y = X \text{ wvr not } Y =$$

$$\begin{align*}
& \text{if } \text{first } Y == 0 \\
& \text{then } X \text{ fby } (\text{next } X \text{ nwvr next } Y) \\
& \text{else } (\text{next } X \text{ nwvr next } Y)
\end{align*}$$

$$X \text{ nwvr } Y = X @ T \text{ where}$$

$$\begin{align*}
T &= U \text{ fby } U @ (T + 1) \\
U &= \text{if } Y == 0 \text{ then } \# \text{ else next } U
\end{align*}$$

$\text{nrwvr}$ stands for *do not retreat whenever*. $\text{nrwvr}$ chooses from its left-hand-side operand backwards only values in the current dimension where the right-hand-side evaluates to $false$.

$$X \text{ nrwvr } Y = X \text{ rwvr not } Y =$$

$$\begin{align*}
& \text{if } \text{last } Y == 0 \\
& \text{then } X \text{ pby } (\text{prev } X \text{ nrwvr prev } Y) \\
& \text{else } (\text{prev } X \text{ nrwrvr prev } Y)
\end{align*}$$
\[
X \ \text{rnwvr} \ Y = X @ T \quad \text{where} \\
T = U \ \text{pby} \ U @ (T - 1) \\
U = \begin{cases} 
0 & \text{if } Y == 0 \\
\text{prev } U & \text{else}
\end{cases}
\]
end

- **asa** stands for *as soon as*. **asa** returns the value of its left-hand-side as a first point in that stream as soon as the right-hand-side evaluates to *true*.

\[
X \ \text{asa} \ Y = \text{first} (X \ \text{wvr} \ Y)
\]

- **ala** (other suggested name is rasas) stands for *as late as* (or *reverse of a soon as*). **ala** returns the value of its left-hand-side as the last point in that stream when the right-hand-side evaluates to *true* for the last time.

\[
X \ \text{ala} \ Y = \text{last} (X \ \text{wvr} \ Y)
\]

- **nasa** stands for *not as soon as*. **nasa** returns the value of its left-hand-side as a first point in that stream as soon as the right-hand-side evaluates to *false*.

\[
X \ \text{nasa} \ Y = \text{first} (X \ \text{nwvr} \ Y)
\]

- **nala** (other suggested name is nrasas) stands for *not as late as* (or *reverse of not a soon as*). **nala** returns the value of its left-hand-side as the last point in that stream when the right-hand-side evaluates to *false* for the last time.

\[
X \ \text{nala} \ Y = \text{last} (X \ \text{nrwvr} \ Y)
\]

- **upon** stands for *advances upon*. Unlike **asa**, **upon** switches context of its left-hand-side operand if the right-hand-side is *true*.

\[
X \ \text{upon} \ Y = X \ \text{fby} ( \\
\text{if } \text{first} \ Y \neq 0 \\
\quad \text{then} \ (\text{next } X \ \text{upon } \text{next} \ Y) \\
\quad \text{else} \ (X \ \text{upon } \text{next} \ Y))
\]

\[
X \ \text{upon} \ Y = X @ W \quad \text{where} \\
W = 0 \ \text{fby} (\text{if } Y \ \text{then} \ (W + 1) \ \text{else} \ W)
\]
end

- **rupon** stands for *retreats upon*. **rupon** switches context backwards of its left-hand-side operand if the right-hand-side is *true*.

\[
X \ \text{rupon} \ Y = X \ \text{pby} ( \\
\text{if } \text{last} \ Y \neq 0 \\
\quad \text{then} \ (\text{prev } X \ \text{rupon } \text{prev} \ Y) \\
\quad \text{else} \ (X \ \text{rupon } \text{prev} \ Y))
\]

\[
X \ \text{rupon} \ Y = X @ W \quad \text{where} \\
W = 0 \ \text{pby} (\text{if } Y \ \text{then} \ (W - 1) \ \text{else} \ W)
\]
end

82
- **NUPON** stands for *not advances upon* or rather *advances otherwise*. **NUPON** switches context of its left-hand-side operand if the right-hand side is *false*.

\[
X \text{ nupon } Y = X \text{ upon not } Y = X \text{ fby (}
\begin{align*}
\text{if } & \text{ first } Y == 0 \\
\text{ then (next } & X \text{ nupon next } Y) \\
\text{ else (} & X \text{ nupon next } Y))
\end{align*}
\]

\[X \text{ nupon } Y = X@W \text{ where } W = 0 \text{ fby (if } Y == 0 \text{ then (} W + 1 \text{ else } W))\text{ end}\]  

(22)

- **NRUPON** stands for *not retreats upon*. **NRUPON** switches context backwards of its left-hand-side operand if the right-hand side is *false*.

\[
X \text{ nrupon } Y = X \text{ rupon not } Y = X \text{ pby (}
\begin{align*}
\text{if } & \text{ last } Y == 0 \\
\text{ then (prev } & X \text{ nrupon prev } Y) \\
\text{ else (} & X \text{ nrupon prev } Y))
\end{align*}
\]

\[X \text{ nrupon } Y = X@W \text{ where } W = 0 \text{ pby (if } Y == 0 \text{ then (} W - 1 \text{ else } W))\text{ end}\]  

(23)

### 2.4 Forensic Operators

The operators presented here are based on the discussion of the combination function and others that form more-than-primitive operations to support the required implementation. The discussed earlier *comb()* operator needs to be realized in the general manner for combining analogies of MPRs, which in our case are higher-level contexts, in the new language’s dimension types.

- **combine** corresponds to the *comb* function as originally described by Gladyshev in [22]. It is defined in Listing 1.2. It is a preliminary context-enhanced version.

```java
/**
 * Append given e to each element
 * of a given stream e under the
 * context of d.
 * @return the resulting combined stream
 */
combine(s, e, d) =
  if iseod s then eod;
  else (first s fby.d e) fby.d combine(next s, e, d);
fi
```

**Listing 1.2. The combine Operator**

- **product** tentatively corresponds to the cross-product [22] of contexts. It is defined in Listing 1.3.

The translated examples show recursion that we are not prepared to deal with in the current Lucid semantics, and will address that in the future work. The two illustrated operators are the first of the a few more to follow in the final language prototype.
2.5 Operational Semantics

As previously mentioned, the operational semantics of Forensic Lucid for the large part is viewed as a composition of the semantic rules of Indexical Lucid, Objective Lucid, and Lucx along with the new operators and definitions. Here we list the existing combined semantic definitions to be used the new language, specifically extracts of operational semantics from GIPL [11], and Lucx [13] are in Figure 1, and Figure 3 respectively. The explanation of the rules and the notation are given in great detail in the cited works and are trimmed in this article. For convenience of the reader they are recited here to a degree. The new rules of the operational semantics of Forensic Lucid cover the newly defined operators primarily, including the reverse and logical stream operators as well as forensic-specific operators. We use the same notation as the referenced languages to maintain consistency in defining our rules.

In the implementing system, GIPSY, the GIPL is the generic counterpart of all the Lucid programming languages. Like Indexical Lucid, which it is derived from, it has only the two standard intensional operators: \( E \odot C \) for evaluating an expression \( E \) in context \( C \), and \#d for determining the position in dimension \( d \) of the current context of evaluation in the context space [11]. SIPLs are Lucid dialects (Specific Intensional Programming Languages) with their own attributes and objectives. Theoretically, all SIPLs can be translated into the GIPL [11]. All the SIPLs conservatively extend the GIPL syntactically and semantically. The remainder of this section presents a relevant piece of Lucx as a conservative extension to GIPL. The semantics of GIPL is presented in Figure 1. The excerpt of semantic rules of Lucx are then presented as a conservative extension to GIPL and Lucx in Figure 3. Following is the description of the GIPL semantic rules as presented in [11]:

\[ \mathcal{D} \vdash E : v \]

tells that under the definition environment \( \mathcal{D} \), expression \( E \) would evaluate to value \( v \).

\[ \mathcal{D}, \mathcal{P} \vdash E : v \]

specifies that in the definition environment \( \mathcal{D} \), and in the evaluation context \( \mathcal{P} \) (sometimes also referred to as a point in the context space), expression \( E \) evaluates to \( v \). The definition environment \( \mathcal{D} \) retains the definitions of all of the identifiers that appear in a Lucid program, as created with the semantic rules 13-16 in Figure 1. It is therefore a partial function

\[ \mathcal{D} : \text{Id} \rightarrow \text{IdEntry} \]

where \text{Id} is the set of all possible identifiers and \text{IdEntry}, has five possible kinds of value, one for each of the kinds of identifier: 1. \text{Dimensions} define the coordinate pairs, in which one can navigate with the \# and \@ operators. Their \text{IdEntry} is simply \((\text{dim})\). 2. \text{Constants} are external entities that provide a single value, regardless of the context of evaluation. Examples are integers and Boolean values. Their \text{IdEntry} is \((\text{const}, c)\), where \( c \) is the value of the constant. 3. \text{Data operators} are external entities that provide memoryless functions. Examples are the arithmetic and Boolean functions. The constants and data operators are said to define the basic algebra of the language. Their \text{IdEntry} is \((\text{op}, f)\), where \( f \) is the function itself. 4. \text{Variables} carry the multidimensional streams. Their \text{IdEntry} is \((\text{var}, E)\), where \( E \) is the Lucid expression defining the variable. It should be noted that this semantics makes the assumption that all variable names are unique. This constraint is easy to overcome by performing compile-time renaming or using a nesting level environment scope when needed. 5. \text{Functions} are non-recursive GIPL user-defined functions. Their \text{IdEntry} is \((\text{func}, \text{id}, E)\), where the \( \text{id} \) are the formal parameters to the function and \( E \) is the body of the function. In this paper we do not discuss the semantics of recursive functions.

Listing 1.3. The product Operator

```c
/
* Append elements of s2 to element of s1
* in all possible combinations.
*
product(s1, s2, d) =
  if iseo2 s2 then eod;
  else combine(s1, first s2) fby.d product(s1, next s2);
fi
```

84
E_{\text{vid}} : \frac{\text{id} = (\text{const,}\ c)}{\mathcal{D}, \mathcal{P} \vdash \text{id} : c} \quad (24)

E_{\text{opid}} : \frac{\text{id} = (\text{op,}\ f)}{\mathcal{D}, \mathcal{P} \vdash \text{id} : \text{id}} \quad (25)

E_{\text{id}} : \frac{\text{id} = (\text{dim})}{\mathcal{D}, \mathcal{P} \vdash \text{id} : \text{id}} \quad (26)

E_{\text{id}} : \frac{\text{id} = (\text{func,}\ id, E)}{\mathcal{D}, \mathcal{P} \vdash \text{id} : \text{id}} \quad (27)

E_{\text{vid}} : \frac{\text{id} = (\text{var,}\ E)}{\mathcal{D}, \mathcal{P} \vdash \text{id} : v} \quad (28)

E_{\text{op}} : \frac{\mathcal{D}, \mathcal{P} \vdash E : \text{id}}{\mathcal{D}, \mathcal{P} \vdash (\text{op,}\ f) : \mathcal{D}, \mathcal{P} \vdash E_i : v_i} \quad (29)

E_{\text{fct}} : \frac{\mathcal{D}, \mathcal{P} \vdash E : \text{id}}{\mathcal{D}, \mathcal{P} \vdash (\text{func,}\ id, E') : \mathcal{D}, \mathcal{P} \vdash \text{id} \leftarrow E_i : v \vdash \mathcal{D}, \mathcal{P} \vdash E(E_1, \ldots, E_n) : f(v_1, \ldots, v_n)} \quad (30)

E_{\text{cr}} : \frac{\mathcal{D}, \mathcal{P} \vdash E : \text{true}}{\mathcal{D}, \mathcal{P} \vdash \text{true} : v'} \quad (31)

E_{\text{eq}} : \frac{\mathcal{D}, \mathcal{P} \vdash E : \text{false}}{\mathcal{D}, \mathcal{P} \vdash \text{false} : v''} \quad (32)

E_{\text{tag}} : \frac{\mathcal{D}, \mathcal{P} \vdash \text{id} : \text{id}}{\mathcal{D}, \mathcal{P} \vdash \#E : \mathcal{D}(\text{id})} \quad (33)

E_{\text{at}} : \frac{\mathcal{D}, \mathcal{P} \vdash E' : \text{id}}{\mathcal{D}, \mathcal{P} \vdash (\text{dim}) : \mathcal{D}, \mathcal{P} \vdash \#E' : \mathcal{D}(\text{id})} \quad (34)

E_{\text{w}} : \frac{\mathcal{D}, \mathcal{P} \vdash Q : \mathcal{D}', \mathcal{P}' \vdash E : v}{\mathcal{D}, \mathcal{P} \vdash E \text{ where } Q : v} \quad (35)

Q_{\text{dim}} : \frac{\mathcal{D}, \mathcal{P} \vdash \text{dimension id} : \mathcal{D} \vdash [\text{id} \mapsto (\text{dim})], \mathcal{P} \vdash [\text{id} \mapsto 0]} {\mathcal{D}, \mathcal{P} \vdash \text{id} : \mathcal{D} \vdash (\text{var,}\ E), \mathcal{P}} \quad (36)

Q_{\text{id}} : \frac{\mathcal{D}, \mathcal{P} \vdash \text{id} : E : \mathcal{D} \vdash [\text{id} \mapsto (\text{var,}\ E)], \mathcal{P}} {\mathcal{D}, \mathcal{P} \vdash \text{id} : \mathcal{D} \vdash (\text{func,}\ id_i, E), \mathcal{P}} \quad (37)

Q_{\text{id}} : \frac{\mathcal{D}, \mathcal{P} \vdash \text{id}(id_1, \ldots, id_n) : E : \mathcal{D} \vdash [\text{id} \mapsto (\text{func,}\ id_i, E)], \mathcal{P}} {\mathcal{D}, \mathcal{P} \vdash \text{id} : \mathcal{D} \vdash (\text{func,}\ id_i, E), \mathcal{P}} \quad (38)

Q_{\text{Q}} : \frac{\mathcal{D}, \mathcal{P} \vdash Q : \mathcal{D}', \mathcal{P}' \vdash Q' : \mathcal{D}'', \mathcal{P}''}{\mathcal{D}, \mathcal{P} \vdash Q' : \mathcal{D} \vdash (\text{fun,}\ id_i, E), \mathcal{P}'} \quad (39)

\begin{align*}
\text{Fig. 1. GIPL Semantics} & \\
E_{\text{E.dim}} & : \frac{\mathcal{D}(E, \text{id}) = (\text{dim})}{\mathcal{D}, \mathcal{P} \vdash \text{id} : \text{id}, \text{id}} \quad (40)
\end{align*}

\begin{align*}
\text{Fig. 2. Higher-Order Context Dot Operator} & \\
\text{The evaluation context } \mathcal{D}, \text{ which is changed when the } & \\
\text{\textit{@} operator is evaluated, or a dimension is declared in a} & \\
\text{\textit{where} clause, associates a } \text{tag}\text{ (i.e. an index) to each relevant dimension. It is, therefore, a partial function} & \\
\mathcal{D} & : \text{Id} \to \mathbb{N}
\end{align*}
Each type of identifiers can only be used in the appropriate situations. Identifiers of type op, func, and dim evaluate to themselves (Figure 1, rules 25,26,27). Constant identifiers (const) evaluate to the corresponding constant (Figure 1, rule 24). Function calls, resolved by the E_{\text{fun}} rule (Figure 1, rule 30), require the renaming of the formal parameters into the actual parameters (as represented by \( E'[\text{id}_1 \mapsto E_i] \)). The function \( \mathcal{P}' = \mathcal{P}'[\text{id} \mapsto v'] \) specifies that \( \mathcal{P}'(x) \) is \( v' \) if \( x = \text{id}_1 \), and \( \mathcal{P}'(x) \) otherwise. The rule for the where clause, \( \mathcal{E}_w \) (Figure 1, rule 35), which corresponds to the syntactic expression \( E \text{ where } Q \), evaluates \( E \) using the definitions \( Q \) therein. The additions to the definition environment \( \mathcal{D} \) and context of evaluation \( \mathcal{P} \) made by the \( Q \) rules (Figure 1, rules 36,37,38) are local to the current where clause. This is represented by the fact that the \( \mathcal{E}_w \) rule returns neither \( \mathcal{D} \) nor \( \mathcal{P} \). The \( \mathcal{Q}_{\text{dim}} \) rule adds a dimension to the definition environment and, as a convention, adds this dimension to the context of evaluation with tag 0 (Figure 1, rule 36). The \( \mathcal{Q}_{\text{id}} \) and \( \mathcal{Q}_{\text{id}} \) simply add variable and function identifiers along with their definition to the definition environment (Figure 1, rules 37,38).

As a conservative extension to GIPL, Lucx’s semantics introduces the notion of context as a building block into the semantic rules, i.e. context as a first-class value, as described by the rules in Figure 3. In Lucx, semantic rule 42 (Figure 3) creates a context as a semantic item and returns it as a context \( \mathcal{P} \) that can then be used by rule 43 to navigate to this context by making it override the current context. GIPL’s semantic rule 29 is still valid for the definition environment \( \mathcal{D} \) and context of evaluation \( \mathcal{P} \) (Figure 3). In Lucx, semantic rule 42 (Figure 3) creates a context as a semantic item and returns it as a context \( \mathcal{P} \) that can then be used by rule 43 to navigate to this context by making it override the current context. GIPL’s semantic rule 29 is still valid for the definition environment \( \mathcal{D} \) and context of evaluation \( \mathcal{P} \) (Figure 3).

\[ E_{\#}^{(\text{ex})} : \quad \mathcal{D}, \mathcal{P} \vdash \# : \mathcal{D} \]

\[ E_{\text{construction}}^{(\text{ex})} : \quad \begin{align*}
\mathcal{D}, \mathcal{P} \vdash E_{d_1} : \text{id}_1 \\
\mathcal{D}, \mathcal{P} \vdash E_{v_j} : v_j
\end{align*} \quad \begin{align*}
\mathcal{D} & = \mathcal{D} \upharpoonright [\text{id}_1 \mapsto v_1] \upharpoonright \ldots \upharpoonright [\text{id}_n \mapsto v_n] \\
\mathcal{P} & \vdash E_{d_1} : E_{i_1}, E_{d_2} : E_{i_2}, \ldots, E_{d_n} : E_{i_n} : \mathcal{P}'
\end{align*}

\[ E_{\text{at}}^{(\text{ex})} : \quad \begin{align*}
\mathcal{D}, \mathcal{P} \vdash E' : \mathcal{P}' \\
\mathcal{D}, \mathcal{P} \vdash E : v
\end{align*} \quad \begin{align*}
\mathcal{D}, \mathcal{P} \vdash E @ E' : v
\end{align*}

Fig. 3. Conservative Semantic Rules Introduced by Lucx

3 Conclusion

While the list of Isabelle’s proofs is incomplete at the time of the writing of this manuscript some formalization in Isabelle took place, and the work on them is currently on-going.

3.1 Results

Due to a non-standard nature of the Lucid language (as opposed to standard imperative languages), it takes some time to understand the full scope of some of its details and model them. This complicates a way to model its operators, expressions, overall meaning in Isabelle. This fact resulted in several trials and attempts to approach the language, from fairly complex to fairly basic – plain integers and pipelined processing and basic index support. They are not fully complete, but some of the basic properties are modeled and proven; please refer to the Isabelle sources for details (once completed it is planned to be released as a part of the Archive of Formal Proofs at [27]).

- The \text{IntegerLucid} Isabelle file is the most developed out of all as far as definition and exploitation of intensional operators of classical Lucid concerned. It is called “integer” because all the streams and dimensions and all operators around them play with integers, natural numbers, and in rarer cases Booleans. There are no identifiers in there. The Isabelle file contains three theories: \text{OriginalLucidOperators}, \text{LucidOperators}, and \text{IntegerLucid}. The first models classical Lucid operators as pipelined dataflows. The second adds up some index support and proves equivalence to the first definitions. The latter provides new definitions of the intensional operators through @ and #, defines meaning functions, propositions, and lemmas from [11]. Integer Lucid proves the example for \( N @.d 2 = 44 \) for the at (\).

86
– The BasicLucid theory is currently the second one derived to support Lucid definitions. It is an extension of IntegerLucid by adding identifiers. asa and upon are in this theory.
– The LucidSemanticRules theory is meant to have the meaning of complete semantic rules and proven, but it only has a definition of a Hoare tuple [28] and a meaning function for it.
– The CommonLucidTypes theory is used by all (most) theories and defines some common types used by most [29].
– ForensicLucid.thy, GIPL.thy, IndexicalLucid.thy, JLucid.thy, J00IP.thy, Lucx.thy, ObjectiveLucid.thy are the theories under current development with some results from the above. The completed work will have a complete list of the files publicly available and submitted to the AfP [27].

3.2 Future Work

The near-future work will consist primarily of the following items:

– Complete semantics of all the mentioned Lucid dialects and their formalization with Isabelle.
– Augment the language specification to include the Depmster-Shafer theory [30,31] of evidence to allow weights for claims, credibility, belief, and plausibility parameters.
– Prove semantic rules involving intensional data warehouse.
– Implementation of the Forensic Lucid compiler, run-time and interactive development environments.

4 Acknowledgments

This research and development work was funded in part by NSERC and the Faculty of Engineering and Computer Science of Concordia University, Montreal, Canada. Thanks to Drs. Mourad Debbabi, Patrice Chalin, Peter Grogono on valuable suggestions used in this work.

References

Appendix

A Lucid Axioms, Theorems, and Proofs

Here we present and extend the notion of the formalisms from Paquet [11] and extend them on to the present work.

A.1 Streaming and Basic Operators

The origins of Lucid date back to 1974. At that time, Ashcroft and Wadge were working on a purely declarative language, in which iterative algorithms could be expressed naturally, which eventually resulted in [9]. Their work fits into the broad area of research into program semantics and verification. It would later turn out that their work is also relevant to the dataflow networks and coroutines of Kahn and MacQueen [32,33]. In the original Lucid (whose
operators are in \textit{this font}, streams were defined in a pipelined manner, with two separate definitions: one for the initial element, and another one for the subsequent elements. For example, the equations

\begin{align*}
\text{first } X &= 0 \\
\text{next } X &= X + 1
\end{align*}

define variable $X$ to be a stream, such that

\begin{align*}
x_0 &= 0 \\
x_{i+1} &= x_i + 1
\end{align*}

In other words,

\begin{align*}
0 &= (0, 0, 0, \ldots, 0, \ldots) \\
X &= (x_0, x_1, \ldots, x_i, \ldots) = (0, 1, \ldots, i, \ldots)
\end{align*}

Similarly, the equations

\begin{align*}
\text{first } X &= X \\
\text{next } Y &= Y + \text{next } X
\end{align*}

define variable $Y$ to be the running sum of $X$, i.e.

\begin{align*}
y_0 &= x_0 \\
y_{i+1} &= y_i + x_{i+1}
\end{align*}

In other words,

\begin{align*}
Y &= (y_0, y_1, \ldots, y_i, \ldots) = \left(0, 1, \ldots, \frac{i(i+1)}{2}, \ldots\right)
\end{align*}

It soon became clear that a “new” operator at the time, \texttt{fby} (followed by) can be used to define such typical situations. Hence, the above two variables could be defined as follows:

\begin{align*}
X &= 0 \texttt{fby } X + 1 \\
Y &= X \texttt{fby } Y + \text{next } X
\end{align*}

As a result, we can summarize the three basic operators of the original Lucid.

\begin{definition}
If $X = (x_0, x_1, \ldots, x_i, \ldots)$ and $Y = (y_0, y_1, \ldots, y_i, \ldots)$, then

\begin{enumerate}
\item $\text{first } X \triangleq (x_0, x_0, \ldots, x_0, \ldots)$
\item $\text{next } X \triangleq (x_1, x_2, \ldots, x_{i+1}, \ldots)$
\item $X \texttt{fby } Y \triangleq (x_0, y_0, y_1, \ldots, y_{i-1}, \ldots)$
\end{enumerate}
\end{definition}

Here parallels can be drawn to the list operations, where \texttt{first} corresponds to \texttt{head}, \texttt{next} corresponds to \texttt{tail}, and \texttt{fby} corresponds to \texttt{cons}. When these operators are combined with Landin’s ISWIM [34] (If You See What I Mean), essentially typed lambda-calculus with syntactic sugar, it becomes possible to define complete Lucid programs. The following three derived operators have turned out to be very useful (we will use them later in the text):

\begin{definition}
\begin{enumerate}
\item $X \texttt{wvr } Y \triangleq \text{if } \text{first } Y \text{ then } X \texttt{fby } (\text{next } X \texttt{wvr next } Y) \text{ else } (\text{next } X \texttt{wvr next } Y)$
\item $X \texttt{asa } Y \triangleq \text{first } (X \texttt{wvr } Y)$
\item $X \texttt{upon } Y \triangleq X \texttt{fby } (\text{if } \text{first } Y \text{ then } (\text{next } X \texttt{upon next } Y) \text{ else } (X \texttt{upon next } Y))$
\end{enumerate}
\end{definition}
Where wvr stands for whenever, asa stands for as soon as and upon stands for advances upon.

A.2 Random Access to Streams

With the original Lucid operators, one could only define programs with pipelined dataflows, i.e. in which the \((i + 1)\)-th element in a stream is only computed once the \(i\)-th element has been computed. This situation is potentially wasteful of resources, since the \(i\)-th element might not necessarily be required. More importantly, it only allows sequential access into streams.

By taking a different approach, it is possible to have random access into streams, using an index \# corresponding to the current position, the current context of evaluation. No longer are we manipulating infinite extensions (streams), rather we are defining computation according to a context (here a single integer). We have set out on the road to intensional programming. We redefine all original Lucid operators in terms of the operators \# and @:

Definition 3

\[
\begin{align*}
(1) & \quad \# & \overset{\text{def}}{=} & 0 \overline{\text{fby}} (# + 1) \\
(2) & \quad X \circ Y & \overset{\text{def}}{=} & \text{if } Y = 0 \text{ then } \text{first } X \\
& & & \text{else } (\text{next } X) \circ (Y \circ 1)
\end{align*}
\]
Further, we give definitions for the original operators using these two baseline operators. In so doing, we will use the following axioms.

**Axiom 1** Let $i \geq 0$.

1. $[c]_i = c$
2. $[X + c]_i = [X]_i + c$
3. $[\text{first } X]_i = [X]_0$
4. $[\text{next } X]_i = [X]_{i+1}$
5. $[X \text{ fby } Y]_0 = [X]_0$
6. $[X \text{ fby } Y]_{i+1} = [Y]_i$
7. $\text{if true then } [X]_i \text{ else } [Y]_i = [X]_i$
8. $\text{if false then } [X]_i \text{ else } [Y]_i = [Y]_i$
9. $[\text{if } C \text{ then } X \text{ else } Y]_i = [\text{if } C]_i \text{ then } [X]_i \text{ else } [Y]_i$

Prior giving the re-definitions of the standard Lucid operators, we show some basic properties of $\otimes$ and $\#$. We will use throughout the discussion here $[X]_i$ instead of $x_i$, as it allows for greater readability. Furthermore, we will, as is standard, write $X = Y$ whenever we have $(\forall i : i \geq 0 : [X]_i = [Y]_i)$

**Proposition 1.** Let $i \geq 0$.

1. $[#]_i = i$
2. $[X \otimes Y]_i = [X]_i [Y]_i$

**Proof**

(1) Proof by induction over $i$.

**Base step** ($i = 0$).

$[#]_0 = [0 \text{ fby } (# + 1)]_0$

$= [0]_0$

$= 0$

**Induction step** ($i = k + 1$). Suppose $(\forall i : i \leq k : [#]_i = i)$.

$[#]_{k+1} = [0 \text{ fby } (# + 1)]_{k+1}$

$= [# + 1]_k$

$= [#]_k + 1$

$= k + 1$

Hence $(\forall i : i \geq 0 : [#]_i = i)$.

(2) Let $i \geq 0$. We will prove by induction over $y_i$ that $y_i \geq 0 \Rightarrow [X \otimes Y]_i = [X]_i [Y]_i$.

**Base step** ($y_i = 0$).

$[X \otimes Y]_i = [\text{if } Y = 0 \text{ then } \text{first } X \text{ else } (\text{next } X) \otimes (Y - 1)]_i$

$= [\text{if } Y = 0]_i \text{ then } [\text{first } X]_i \text{ else } [(\text{next } X) \otimes (Y - 1)]_i$

$= [\text{if } Y]_i = 0 \text{ then } [\text{first } X]_i \text{ else } [(\text{next } X) \otimes (Y - 1)]_i$

$= [\text{first } X]_i$

$= [X]_0$

$= [X]_i$
**Induction step** \((y_i = k + 1)\). Suppose \((\forall i : i \leq k : \#i = i)\).

\[
[X \otimes Y]_i = \begin{cases} 
  \text{first } X & \text{if } Y = 0 \\
  \text{next } X @ (Y - 1) & \text{if } Y > 0 
\end{cases} 
\]

Defn. 3.2

\[
= \text{if } Y = 0 \text{ then } [\text{first } X]_i \text{ else } ([\text{next } X] \otimes (Y - 1))_i 
\]

Axiom 1.9

\[
= \text{if } Y = 0 \text{ then } [\text{first } X]_i \text{ else } ([\text{next } X] \otimes (Y - 1))_i 
\]

Axiom 1.2

\[
= ([\text{next } X] \otimes (Y - 1))_i 
\]

Axiom 1.8

Ind. Hyp.

\[
= [\text{next } X]_{y_i - 1} 
\]

Axiom 1.2

\[
= [X]_{y_i - 1} + 1 
\]

Axiom 1.4

\[
= [X]_{y_i} + 1 
\]

Arith.

Hence \((\forall i : i \geq 0 : Y_i \geq 0 \Rightarrow ([X \otimes Y]_i = [X]_{y_i}))\).

**Definition 4**

1. \(\text{first } X \overset{\text{def}}{=} X \otimes 0\)
2. \(\text{next } X \overset{\text{def}}{=} X \otimes (# + 1)\)
3. \(X \text{ fby } Y \overset{\text{def}}{=} \text{if } # = 0 \text{ then } X \text{ else } Y @ (# - 1)\)
4. \(X \text{ wvr } Y \overset{\text{def}}{=} X @ T\)
   \[\text{where}\]
   \(T = U \text{ fby } U @ (T + 1)\)
   \(U = \text{if } Y \text{ then } # \text{ else } \text{next } U\)
   \[\text{end}\]
5. \(X \text{ asa } Y \overset{\text{def}}{=} \text{first } (X \text{ wvr } Y)\)
6. \(X \text{ upon } Y \overset{\text{def}}{=} X @ W\)
   \[\text{where}\]
   \(W = 0 \text{ fby } \text{if } Y \text{ then } (W + 1) \text{ else } W\)
   \[\text{end}\]

The advantage of these new definitions is that they do not use any form of recursive function definitions. Rather, all of the definitions are iterative, and in practice, more easily implemented in an efficient manner. We prove below that the new definitions are equivalent to the old ones.

**Proposition 2.** \(\text{first } X = \text{first } X\).

**Proof.** Let \(i \geq 0\). Then

\[
[\text{first } X]_i = [X \otimes 0]_i 
\]

Defn. 4.1

\[
= [X]_0 
\]

Prop. 1.2

\[
= [X]_0 
\]

Axiom 1.1

\[
= [\text{first } X]_i 
\]

Axiom 1.3

Hence \(\text{first } X = \text{first } X\).
Proposition 3. \( \text{next } X = \text{next } X \).

Proof Let \( i \geq 0 \). Then
\[
[\text{next } X]_i = [X \oplus (# + 1)]_i
\]
\[
= [X]_{# + 1,i}
\]
\[
= [X]_{#,i + 1}
\]
\[
= [\text{next } X]_i
\]
Defn. 4.2
Prop. 1.2
Axiom 1.4

Hence \( \text{next } X = \text{next } X \).

Proposition 4. \( X fby Y = X fby Y \).

Proof Proof by induction over \( i \).

Base step \((i = 0)\).
\[
[X fby Y]_0 = [\text{if } # = 0 \text{ then } X \text{ else } Y \oplus (# - 1)]_0
\]
\[
= [\text{if } # = 0]_0 \text{ then } [X]_0 \text{ else } [Y \oplus (# - 1)]_0
\]
\[
= [\text{if } 0]_0 \text{ then } [X]_0 \text{ else } [Y \oplus (# - 1)]_0
\]
\[
= [X]_0
\]
\[
= [X fby Y]_0
\]
Defn. 4.3
Axiom 1.9
Prop. 1.1
Axiom 1.7

Induction step \((i = k + 1)\).
\[
[X fby Y]_{k+1} = [\text{if } # = 0 \text{ then } X \text{ else } Y \oplus (# - 1)]_{k+1}
\]
\[
= [\text{if } # = 0]_{k+1} \text{ then } [X]_{k+1} \text{ else } [Y \oplus (# - 1)]_{k+1}
\]
\[
= [\text{if } 0]_{k+1} \text{ then } [X]_{k+1} \text{ else } [Y \oplus (# - 1)]_{k+1}
\]
\[
= [Y \oplus (# - 1)]_{k+1}
\]
\[
= [Y]_{#-1,k+1}
\]
\[
= [Y]_{#,k+1-1}
\]
\[
= [Y]_{k}
\]
\[
= [X fby Y]_{k+1}
\]
Defn. 4.3
Axiom 1.9
Prop. 1.1
Axiom 1.2
Prop. 1.1

Hence \( \forall i : i \geq 0 : [X fby Y]_i = [X fby Y]_i \). Hence \( fby = fby \).

The proof for \( wvr \) is more complicated, as it requires relating an iterative definition to a recursive definition. We will therefore need four lemmas that refer to variables \( T \) and \( U \) in the text in Definitions 4.4.1 and 4.4.2. In addition, we must define the rank of a Boolean stream. Finally, we will have to introduce another set of axioms, that allow us to compare two entire streams, as opposed to particular elements in the two streams.
Axiom 2  Let $i \geq 0$.

1. $X^0 = X$
2. $[X]^0 = [X]_i$
3. next $X^i = X^{i+1}$
4. first $X^i = X^i$
5. next $(X \text{ fby } Y) = Y$
6. (first $X$) fby $Y = X$ fby $Y$
7. if true then $X$ else $Y = X$
8. if false then $X$ else $Y = Y$

Definition 5  Let $Y$ be a Boolean stream.

1. rank $(-1, Y) \overset{\text{def}}{=} -1$
2. rank $(i + 1, Y) \overset{\text{def}}{=} \min\{k : k > \text{rank}(i, Y) : [Y]_k = \text{true}\}$

Further, we write $r_i$ for rank $(i, Y)$.

Lemma 1. $\forall i : i \geq -1 : (\forall j : r_i < j \leq r_{i+1} : X^j \text{ wvr } Y^j = X^{r_{i+1}} \text{ wvr } Y^{r_{i+1}})$. 

Proof  Let $i \geq -1$. Proof by downwards induction over $j$. Note that $r_i < r_{i+1}$.

Base step ($j = r_{i+1}$).

$$X^{r_{i+1}} \text{ wvr } Y^{r_{i+1}} = X^{r_{i+1}} \text{ wvr } Y^{r_{i+1}} \quad \text{Identity}$$

Induction step ($j = k - 1, j > r_i$).

$$X^{k-1} \text{ wvr } Y^{k-1} = \begin{cases} \text{if first } Y^{k-1} \text{ then } X^{k-1} \text{ fby } X^k \text{ wvr } Y^k & \text{Defn. 2.1} \\ \text{else } X^k \text{ wvr } Y^k & \text{Axiom 2.3} \end{cases}$$

$$= \begin{cases} \text{if } [Y]_{k-1} \text{ then } X^{k-1} \text{ fby } X^k \text{ wvr } Y^k & \text{Axiom 2.8} \\ \text{else } X^k \text{ wvr } Y^k & \text{Ind. Hyp.} \end{cases}$$

Hence, $\forall i : i \geq -1 : (\forall j : r_i < j \leq r_{i+1} : X^j \text{ wvr } Y^j = X^{r_{i+1}} \text{ wvr } Y^{r_{i+1}})$. \hfill \Box$

Lemma 2. $\forall i : i \geq 0 : (X \text{ wvr } Y)^i = X^r \text{ wvr } Y^r$.

Proof  Proof by induction over $i$.

Base step ($i = 0$).

$$(X \text{ wvr } Y)^0 = X \text{ wvr } Y$$
$$= X^0 \text{ wvr } Y^0 \quad \text{Axiom 2.1}$$
$$= X^0 \text{ wvr } Y^0 \quad \text{Lemma 1}$$
Induction step \((i = k + 1)\).

\[
(X \text{ wvr } Y)^{k+1} = \text{next } ((X \text{ wvr } Y)^k) \quad \text{Axiom 2.4}
\]
\[
= \text{next } (X^r \text{ wvr } Y^r) \quad \text{Ind. Hyp.}
\]
\[
= \text{next } (\text{if } \text{first } Y^r \text{ then } X^r \text{ fby } X^{r+1} \text{ wvr } Y^{r+1}
\text{ else } X^{r+1} \text{ wvr } Y^{r+1}) \quad \text{Defn. 2.1}
\]
\[
= \text{next } (\text{if } [Y]_r \text{ then } X^r \text{ fby } X^{r+1} \text{ wvr } Y^{r+1}
\text{ else } X^{r+1} \text{ wvr } Y^{r+1}) \quad \text{Axiom 2.3}
\]
\[
= X^{r+1} \text{ wvr } Y^{r+1} \quad \text{Axiom 2.7}
\]
\[
= X^r \text{ wvr } Y^r \quad \text{Lemma 1}
\]

Hence, \((\forall i : i \geq 0 : (X \text{ wvr } Y)^i = X^r \text{ wvr } Y^r)). \square

Lemma 3. \((\forall i : i \geq -1 : (\forall j : r_i < j \leq r_{i+1} : [U]_j = r_{i+1})))\).

Proof Let \(i \geq -1\). Proof by downwards induction over \(j\). Note that \(r_i < r_{i+1}\).

**Base step** \((j = r_{i+1})\).

\[
[U]_{r_{i+1}} = [\text{if } Y \text{ then } \# \text{ else } \text{next } U]_{r_{i+1}} \quad \text{Defn. 4.4.2}
\]
\[
= \text{if } [Y]_{r_{i+1}} \text{ then } #[r_{i+1}] \text{ else } [\text{next } U]_{r_{i+1}} \quad \text{Axiom 1.9}
\]
\[
= #[r_{i+1}] \quad \text{Axiom 1.7}
\]
\[
= r_{i+1} \quad \text{Prop. 1.1}
\]

**Induction step** \((j = k - 1, j > r_i)\).

\[
[U]_{k-1} = [\text{if } Y \text{ then } \# \text{ else } \text{next } U]_{k-1} \quad \text{Defn. 4.4.2}
\]
\[
= \text{if } [Y]_{k-1} \text{ then } #[k-1] \text{ else } [\text{next } U]_{k-1} \quad \text{Axiom 1.9}
\]
\[
= [\text{next } U]_{k-1} \quad \text{Axiom 1.8}
\]
\[
= [U]_k \quad \text{Axiom 1.4}
\]
\[
= r_{i+1} \quad \text{Ind. Hyp.}
\]

Hence, \((\forall i : i \geq -1 : (\forall j : r_{i-1} < j < r_i : [U]_j = r_{i+1}))). \square
Lemma 4. \((\forall i : i \geq 0 : [T]_i = r_i)\).

**Proof** Proof by induction over \(i\).

**Base step** \((i = 0)\).

\[
[T]_0 = [U \text{ fby } U \otimes (T + 1)]_0
\]

Defn. 4.4.1

\[
= [U]_0
\]

Axiom 1.5

\[
= r_0
\]

Lemma 3

**Induction step** \((i = k + 1)\).

\[
[T]_{k+1} = [U \text{ fby } U \otimes (T + 1)]_{k+1}
\]

Defn. 4.4.1

\[
= [U \otimes (T + 1)]_k
\]

Axiom 1.6

\[
= [U]_{k+1}
\]

Prop. 1.2

\[
= [U]_{k+1}
\]

Ind. Hyp.

\[
= [U]_{k+1}
\]

Lemma 3

\[
= r_{k+1}
\]

Hence, \((\forall i : i \geq 0 : [T]_i = r_i)\). \(\square\)

---

**Proposition 5.** \(X \wvr Y = X \wvr Y\).

**Proof**

\[
[X \wvr Y]_i = [X \otimes T]_i
\]

Defn. 4.4

\[
= [X]_{[T]_i}
\]

Prop. 1.2

\[
= [X]_{r_i}
\]

Lemma 4

\[
= [X]_{r_i}
\]

Defn. 4.4.1

\[
= [X]_{r_i}
\]

Axiom 1.2

\[
= \text{if } [Y]_{r_i} \text{ then } X_{r_i} \text{ fby } X_{r_i+1} \wvr Y_{r_i+1}\]
\[
\quad \text{else } X_{r_i+1} \wvr Y_{r_i+1}]
\]

Axiom 1.6

\[
= \text{if FIRST } Y_{r_i} \text{ then } X_{r_i} \text{ fby } X_{r_i+1} \wvr Y_{r_i+1}\]
\[
\quad \text{else } X_{r_i+1} \wvr Y_{r_i+1}]_0
\]

Axiom 2.7

\[
= [X_{r_i} \wvr Y_{r_i}]_0
\]

Defn. 2.1

\[
= (X \wvr Y)_{[i]}_0
\]

Lemma 2

\[
= [X \wvr Y]_{[i]}
\]

Axiom 2.2

Hence \(X \wvr Y = X \wvr Y\). \(\square\)

---

**Proposition 6.** \(X \asa Y = X \asa Y\).

**Proof**

\[
X \asa Y = \text{first } (X \wvr Y)
\]

Defn. 4.5

\[
= \text{first } (X \wvr Y)
\]

Prop. 5

\[
= \text{FIRST } (X \wvr Y)
\]

Prop. 2

\[
= X \asa Y
\]

Defn. 2.2

Hence \(X \asa Y = X \asa Y\). \(\square\)
Lemma 5. \(\forall i : i \geq 0 : (X \text{ upon } Y)^i = X^[W]^i \text{ upon } Y^i\)

Proof
Proof by induction over \(i\).

Base step \((i = 0)\).
\[ (X \text{ upon } Y)^0 = X \text{ upon } Y \]
\[ = X^0 \text{ upon } Y^0 \quad \text{Axiom 2.1} \]
\[ = X^{(0 \text{ fby } \ldots)}_0 \text{ upon } Y^0 \quad \text{Defn. 2.3} \]
\[ = X^[W]^0 \text{ upon } Y^0 \quad \text{Defn. 4.6.1} \]

Induction step \((i = k + 1)\).
\[ (X \text{ upon } Y)^{k+1} = \text{next} \big((X \text{ upon } Y)^k\big) \]
\[ = \text{next} \big(X^[W]^k \text{ upon } Y^k\big) \]
\[ = \text{if} \ (\text{first } Y^k) \]
\[ \quad \text{then} \big(X^[W]^{k+1} \text{ upon } Y^{k+1}\big) \]
\[ \quad \text{else} \big(X^[W]^k \text{ upon } Y^{k+1}\big) \]
\[ = \text{if} \ [Y]_k \]
\[ \quad \text{then} \big(X^[W]^{k+1} \text{ upon } Y^{k+1}\big) \]
\[ \quad \text{else} \big(X^[W]^k \text{ upon } Y^{k+1}\big) \]
\[ = X^[W]^{k+1} \text{ upon } Y^{k+1} \quad \text{Defn. 4.6.1} \]

Hence, \(\forall i : i \geq 0 : (X \text{ upon } Y)^i = X^[W]^i \text{ upon } Y^i\) \(\square\)

Proposition 7. \(X \text{ upon } Y = X \text{ upon } Y\).

Proof
Let \(i \geq 0\). Then
\[ [X \text{ upon } Y]_i = [X \@ W]_i \quad \text{Defn. 4.6} \]
\[ = [X]_i^[W]_i \quad \text{Prop. 1.2} \]
\[ = [X^[W]_i]^0 \quad \text{Axiom 2.2} \]
\[ = [X^[W] \text{ fby } \ldots]^0 \]
\[ = [X^[W] \text{ upon } Y]^0 \]
\[ = [X \text{ upon } Y]_i \quad \text{Defn. 2.3} \]

Hence \(X \text{ upon } Y = X \text{ upon } Y\). \(\square\)

Now that the corresponding definitions are shown to be equivalent, we can generalize and head off in the negative direction as well:

Definition 6

1. \(\text{prev } X \overset{\text{def}}{=} X \@ (# - 1)\)
2. \(X \text{ fby } Y \overset{\text{def}}{=\ if \ # \leq 0 \ then \ X \ else \ Y \@ (# - 1)}\)
B Summary of the Operators’ Examples

Here we illustrate a few basic examples of application of the Forensic Lucid operators (both, classical Lucid and the newly introduced operators). Assume we have two bounded (between bod and eod) streams X and Y of ten elements. The X stream is just an ordered sequence of natural numbers between 1 and 10. If queried for values below 1 an beginning-of-data (bod) marker would be returned; similarly if queried beyond 10, the end-of-data marker (eod) is returned. The Y stream is a sequence of ten truth values (can be replaced with 0 for “false” and 1 for “true”). The operators applied to these streams may return bounded or unbounded streams of the same or different length than the original depending on the definition of a particular operator. Also assume the current dimension index is 0. The resulting table showing the application of the classical and the new operators is in Table 1.

<table>
<thead>
<tr>
<th>Stream/Index</th>
<th>X (stream)</th>
<th>Y (stream)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>bod</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>T</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>T</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>eod</td>
</tr>
<tr>
<td>10</td>
<td>eod</td>
<td>eod</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stream/Index</th>
<th>X (stream)</th>
<th>Y (stream)</th>
</tr>
</thead>
<tbody>
<tr>
<td>X FIRST Y</td>
<td>1 1 1 1 1 1 1 1 1 1</td>
<td>T T T T T T T T T T</td>
</tr>
<tr>
<td>X LAST Y</td>
<td>10 10 10 10 10 10 10 10 10 10</td>
<td></td>
</tr>
<tr>
<td>X NEXT Y</td>
<td>2 3 4 5 6 7 8 9 eod eod</td>
<td></td>
</tr>
<tr>
<td>X PREV Y</td>
<td>bod</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stream/Index</th>
<th>X (stream)</th>
<th>Y (stream)</th>
</tr>
</thead>
<tbody>
<tr>
<td>X FBY Y</td>
<td>1 T T T T T</td>
<td></td>
</tr>
<tr>
<td>X PBY Y</td>
<td>T F T T T T T</td>
<td></td>
</tr>
<tr>
<td>X WVR Y</td>
<td>2 3 5 6 9</td>
<td></td>
</tr>
<tr>
<td>X RWVR Y</td>
<td>9 6 5 3 2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stream/Index</th>
<th>X (stream)</th>
<th>Y (stream)</th>
</tr>
</thead>
<tbody>
<tr>
<td>X ASA Y</td>
<td>1 1 1 1 1 1 1 1 1 1</td>
<td>T T T T T T T T T T</td>
</tr>
<tr>
<td>X NASA Y</td>
<td>2 2 2 2 2 2 2 2 2 2</td>
<td></td>
</tr>
<tr>
<td>X ALA Y</td>
<td>10 10 10 10 10 10 10 10 10 10</td>
<td></td>
</tr>
<tr>
<td>X NALA Y</td>
<td>9 9 9 9 9 9 9 9 9 9</td>
<td></td>
</tr>
<tr>
<td>X UPON Y</td>
<td>1 2 2 2 3 3 3 4 5 5 eod</td>
<td></td>
</tr>
<tr>
<td>X RUPON Y</td>
<td>10 9 9 8 7 7 7 6 6 6</td>
<td></td>
</tr>
<tr>
<td>X NUPON Y</td>
<td>1 1 2 3 3 4 5 5 5 6 6 eod</td>
<td></td>
</tr>
<tr>
<td>X NRUPON Y</td>
<td>10 10 9 9 9 8 7 7 6 5 5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stream/Index</th>
<th>X (stream)</th>
<th>Y (stream)</th>
</tr>
</thead>
<tbody>
<tr>
<td>neg X</td>
<td>-1 -2 -3 -4 -5 -6 -7 -8 -9 -10</td>
<td>eod eod</td>
</tr>
<tr>
<td>not Y</td>
<td>T T T F F F F F F F</td>
<td></td>
</tr>
<tr>
<td>X AND Y</td>
<td>1 0 1 0 0 1 1 0 1 eod eod</td>
<td></td>
</tr>
<tr>
<td>X OR Y</td>
<td>1 2 3 5 5 6 7 9 9 11 eod eod</td>
<td></td>
</tr>
<tr>
<td>X XOR Y</td>
<td>0 2 3 5 5 6 9 9 11 eod eod</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Example of Application of Forensic Lucid Operators to Bounded Streams
Verification of LISP Interpreters

Magnus O. Myreen

Computer Laboratory, University of Cambridge, Cambridge, UK magnus.myreen@cl.cam.ac.uk

Abstract. LISP interpreters provide a clean abstraction over machine details such as bounded arithmetic and array-like memory. We describe our approach and partial results in verifying LISP interpreters implemented in ARM, PowerPC and x86 machine code.

1 Introduction

The aim of this project is to create verified machine-code implementations of interpreters for a language similar to LISP 1.5 [4]. The target is to deliver verified LISP interpreters in ARM, PowerPC and x86 machine code. Benefits:

1. LISP interpreters provide a clean abstraction of mathematically unclean machine specific details (boundedness of arithmetic, array-like memory etc.). Hence future program proofs can assume a clean abstraction.
2. Subsets of LISP might turn out to be good targets for compilers from proof tools such as HOL4, Isabelle and Coq.
3. Verified LISP interpreters can serve as a proved-to-be-correct environments for evaluating ACL2 s-expressions.

The fact that the proof goes down to detailed models of commercial machine languages distinguishes our project from the VLISP project [1] which stopped at the algorithm level. All proofs are carried out within HOL4 [7].

2 Approach

Our three-stage approach for creating verified machine-code implementations of LISP interpreters:

a. write efficient machine-code implementations of primitive operations over s-expressions held in a garbage collected heap (cons, car, cdr, eq, + etc.).

b. verify primitive heap operations with aid of automatic decompilation into recursive HOL functions (a form of automatic reverse engineering).

c. automatically plug together verified elementary operations to create verified LISP interpreters (a form of proof-producing compilation).

The key research challenge lies in steps b and c. The solutions we are investigating are centered around a new approach for proving (fully-automatically) that the functional behavior of machine code is described by a recursive function. An illustration of this proof technique is given in the next section. Subsequent sections describe its applications to automatic proof-producing decompilation and compilation. Our approach to decompilation is detailed in Myreen et al. [6]. The compilation work is a result of collaboration with Li, Slind and Owens [2].
2.1 Proving that a recursive function is executed by machine code

We illustrate our proof technique with an example. The illustrated technique can be applied one loop at a time. Subsequent proofs can use previously proved specification, hence nested loops, procedure calls and even some non-properly nested loops can be handled.

Consider the following ARM code, which repeatedly subtracts 10 from register 1 until its (unsigned) value is less than 10,

```
L: cmp r1,#10  \(\) \text{compare } r1 \text{ with } 10
subcs r1,r1,#10 \(\) \text{subtract 10 from } r1, \text{ if } cmp \text{ gave } 10 \leq r1
bcs L \(\) \text{jump to top, if } cmp \text{ gave } 10 \leq r1
```

The following function describes the behavior of the above ARM code.

\[
f(x) = \begin{cases} f(x-10) & \text{if } 10 \leq x \\ x & \text{else} \end{cases}
\]

The correspondence between \(f\) and the ARM code (call it \(\text{code}\)) can be stated as follows. Let \(\text{pc}\ p\) assert that the program counter has value \(p\) and similarly let \(r1\ x\) state that register 1 has value \(x\).

\[
\{ r1 \ x \ast \text{pc} \ p \} \ \text{code} \ \{ r1 \ f(x) \ast \text{pc} \ (p+3) \}
\]

Informally this machine-code specification [5, 6] states: given a state where register 1 has value \(x\) and the program counter is \(p\), \(\text{code}\) will reach a state where register 1 has value \(f(x)\) and the program counter is \(p+3\).

Such a theorem can be proved automatically in two steps: first compose specifications for the individual instructions; and then instantiate a special loop rule. The result of composing the specifications of the individual instructions produces two specifications:

\[
10 \leq x \Rightarrow \{ r1 \ x \ast \text{pc} \ p \} \ \text{code} \ \{ r1 \ (x-10) \ast \text{pc} \ p \}
\]

\[
x < 10 \Rightarrow \{ r1 \ x \ast \text{pc} \ p \} \ \text{code} \ \{ r1 \ x \ast \text{pc} \ (p+3) \}
\]

The desired specification (2) is a consequence of the following loop rule instantiated with \(\text{res} = \lambda x. \ r1 \ x \ast \text{pc} \ p\) and \(\text{res}' = \lambda x. \ r1 \ x \ast \text{pc} \ (p+3)\):

\[
\forall \text{res} \ \text{res}' \ c. \ (\forall x. \ G(x) \Rightarrow \{ \text{res} \ x \} \ c \ \{ \text{res} \ F(x) \}) \land \\
(\forall x. \ \neg G(x) \Rightarrow \{ \text{res} \ x \} \ c \ \{ \text{res}' \ D(x) \}) \Rightarrow \\
(\forall x. \ \text{pre}(x) \Rightarrow \{ \text{res} \ x \} \ c \ \{ \text{res}' \ \text{tailrec}(x) \})
\]

Here \(\text{tailrec}\) is the generic form of any tail-recursion and \(\text{pre}\) ensures termination:

\[
\text{tailrec}(x) = \text{if } G(x) \text{ then } \text{tailrec}(F(x)) \text{ else } D(x)
\]

\[
\text{pre}(x) = \text{if } G(x) \text{ then } \text{pre}(F(x)) \text{ else } \text{true}
\]

tailrec was defined using a trick by Moore and Manolios [3] and \(\text{pre}\) is defined as \(\text{pre}(x) = \exists n. \ \neg G(\text{step}(n,F,x))\) with \(\text{step}\) defined recursively as:

\[
\text{step}(0,F,x) = x \\
\text{step}(n+1,F,x) = \text{step}(n,F,F(x))
\]

The proof of the loop rule above is described in Myreen et al. [6].

2.2 Automatic reverse engineering – decompiling into HOL

Machine code is a sequence of machine words, e.g. the ARM program used in the section above is the following in hexadecimal encoding:

```
E351000A 2241100A 2AFFFFFC
```

When decompiling into HOL, we start with a sequence of machine words representing the code. The goal is to construct a HOL function that describes the effect of the code, and to prove automatically that the HOL function corresponds to the machine code. Decompilation can be fully automated as follows:
1. Automatically derive basic specifications for each individual instruction.
2. Analyse the specifications and from them generate a tail-recursive function describing the code, e.g. the following is generated for the code above:

\[ f(r_1) = \begin{cases} 10 \leq r_1 & \text{then } f(r_1 - 10) \text{ else } r_1 \end{cases} \]

3. Run the proof procedure described above to prove that \( f \) is executed by the original machine code.

### 2.3 Automatic proof-producing compilation

Given a function \( f \) that operates over machine words, e.g.

\[ f(x) = \begin{cases} x < 10 & \text{then } x \text{ else } f(x - 10) \end{cases} \]

we can construct (and automatically prove it equivalent to) a function \( g \) where variable names, let-expressions and if-statements correspond to machine code.

\[ g(r_1) = \begin{cases} r_1 < 10 & \text{then } r_1 \text{ else } \\
\begin{array}{l}
\text{let } r_1 = r_1 \text{ else } \\
\text{in } g(r_1)
\end{array} \]

Such functions are easy to turn into assembly code, which can then be turned into machine code by off-the-shelf assemblers. The assembler-generated machine code is proved correct by the automation described in Section 2.1.

### 3 Resulting specification

The proof automation described above is based on automatically composing specifications to construct correctness theorems. By default, the initial specifications are automatically derived specification for individual machine instructions. However, the user can instead supply the compiler/decompiler with alternative specification in order to build on previously proved specifications.

We make use of this feature for implementation of LISP interpreters. First an abstract data-type was defined for s-expressions: \( \text{Dot } x y \) is a pair, \( \text{Num } i \) is an unbounded integer \( i \), and \( \text{Str } s \) is a character string \( s \). Some basic operations:

\[
\begin{align*}
\text{car} (\text{Dot } x y) &= x \\
\text{cdr} (\text{Dot } x y) &= y \\
\text{ltype} (\text{Num } w) &= \text{Num } 0 \\
\text{ltype} (\text{Str } s) &= \text{Num } 1 \\
\text{ltype} (\text{Dot } x y) &= \text{Num } 2 \\
\text{size} (\text{Num } w) &= 0 \\
\text{size} (\text{Str } s) &= 0 \\
\text{size} (\text{Dot } x y) &= 1 + \text{size } x + \text{size } y
\end{align*}
\]

A new resource specification \( \text{heap} \) was defined which relates variables \( \text{task}, \text{exp}, x, y, s, \text{env} \) of the new type to concrete memory representations using a coupling invariant \( \text{machine}\_\text{repr} \). Here \( a \) is the address of the heap and \( l \) is its capacity.

\[
\begin{align*}
\text{heap} (a, l) (\text{task}, \text{exp}, x, y, s, \text{env}) &= \\
& \exists r_3 r_4 r_5 r_6 r_7 r_8 f. \\
& r_3 \ast r_3 \ast r_4 \ast r_4 \ast r_5 \ast r_5 \ast r_6 \ast r_6 \ast r_7 \ast r_7 \ast r_8 \ast r_8 \ast r_9 \ast r_9 \ast \text{memory } f \ast \\
& \text{machine}\_\text{repr} (\text{task}, \text{exp}, x, y, s, \text{env}, l) (r_3, r_4, r_5, r_6, r_7, r_8, r_9, a, f)
\end{align*}
\]

101
Machine-code implementations for basic LISP operations were verified using decompilation followed by some manual proofs, e.g. ARM code for car of exp:

\[
\begin{align*}
\text{let } \text{exp} = \text{Num} & \Rightarrow \\
\{ \text{heap} (a, l) (\text{task}, \text{exp}, x, y, s, \text{env}) * p \} \\
p : \text{E5933000} \\
\{ \text{heap} (a, l) (\text{task}, (\text{car} \text{exp}), x, y, s, \text{env}) * p (p+1) \}
\end{align*}
\]

A memory allocator with a built-in Cheney-collector was used to implement creation of a new pair Dot \( x \ y \).

The precondition of this operation requires the heap to have enough space to accommodate a new cons-cell.

\[
\begin{align*}
\text{let } \text{exp} = \text{Num} & \Rightarrow \\
\{ \text{heap} (a, l) (\text{task}, \text{exp}, x, y, s, \text{env}) * p \} \\
p : \text{E5933000} \\
\{ \text{heap} (a, l) (\text{task}, (\text{car} \text{exp}), x, y, s, \text{env}) * p (p+1) \}
\end{align*}
\]

When the above specifications were supplied to the compiler it knows what machine code to generate for two new commands: one for calculating \text{car} of \text{exp}

\[
\text{let } \text{exp} = \text{car} \text{exp} \text{ in}
\]

and one for producing a new Dot-pair:

\[
\text{let } \text{exp} = \text{Dot} \ x \ y \text{ in}
\]

Once the compilers language has been extended with sufficiently many such primitive operations, a LISP interpreter can be compiled using the compilation technique from above, since our compilation approach essentially only plugs together previously proved specifications. The top-level specification function defining a simple LISP interpreter \text{lisp eval} is sketched in Figure 1 (on the following page). The correctness theorem will be of the form:

\[
\text{lisp eval pre}(\text{task}, \text{exp}, x, y, s, \text{env}) \Rightarrow \\
\{ \text{heap} (a, l) (\text{task}, \text{exp}, x, y, s, \text{env}) * p \} \\
p : \text{... code...} \\
\{ \text{heap} (a, l) (\text{lisp eval}(\text{task}, \text{exp}, x, y, s, \text{env})) * p (p+\text{code length}) \}
\]

Here \text{lisp eval pre} is an automatically generated predicate which has collected the various side-conditions that need to be true for proper execution of the machine-code implementations, i.e. \text{lisp eval pre} is a function which returns true only if the input makes \text{lisp eval} terminate and the capacity of the heap is not exceeded during execution for this input.

**Acknowledgments.** The author would like to thank Mike Gordon, Konrad Slind, Thomas Tuerk, Anthony Fox, Larry Paulson and John Regehr for research discussions, and is grateful for funding from Osk.Huttusen säätö, Finland, and EPSRC, UK.

**References**

TASK_EVAL = NUM 0
TASK_CONT = NUM 1

LISP_EVAL0 (task, exp, x, y, s, env) =
if x = STR "'" then
  let task = TASK_CONT in
  (task, exp, x, y, s, env)
else
  ...

LISP_EVAL1 (task, exp, x, y, s, env) = ...
LISP_EVAL2 (task, exp, x, y, s, env) = ...
LISP_LOOKUP (exp, x, y, env) = ...
LISP_POP (task, x, y, s) = ...

LISP_EVAL (task, exp, x, y, s, env) =
if task = TASK_EVAL then
  let task = TASK_CONT in
  let x = LTYPE exp in
  if x = NUM 0 then (* exp is NUM *)
    LISP_EVAL (task, exp, x, y, s, env)
  else if x = NUM 1 then (* exp is STR *)
    let (exp, x, y, env) = LISP_LOOKUP (exp, x, y, env) in
      LISP_EVAL (task, exp, x, y, s, env)
  else (* if x = NUM 2 then, *) (* exp is DOT *)
    let (x, exp) = (CAR exp, CDR exp) in
    let (exp, y) = (CAR exp, CDR exp) in
    let (task, exp, x, y, s, env) = LISP_EVAL0 (task, exp, x, y, s, env) in
    LISP_EVAL (task, exp, x, y, s, env)
else (* if task = TASK_CONT then *)
  let (task, x, y, s) = LISP_POP (task, x, y, s) in
  if x = STR "nil" then (* evaluation complete, exit *)
    (task, exp, x, y, s, env)
  else
    if task = STR "nil" then (* one arg has been evaluated *)
      let (task, exp, x, y, s, env) = LISP_EVAL1 (task, exp, x, y, s, env) in
      LISP_EVAL (task, exp, x, y, s, env)
    else (* two args have been evaluated *)
      let (task, exp, x, y, s, env) = LISP_EVAL2 (task, exp, x, y, s, env) in
      LISP_EVAL (task, exp, x, y, s, env)

Fig. 1. Parts of the definition of lisp_eval in HOL4.
Abstract. This approach introduces a coupling of a specification framework with a verification system. Given a system, represented in a formal specification framework, one can verify its properties by translating the specification to a Higher-Order Logic and subsequently using the theorem prover Isabelle/HOL or the point of disagreement will be found. Moreover, using this approach one can validate the refinement relation between two given systems, as well as make automatic correctness proofs of syntactic interfaces for specified system components. The approach uses particularly the idea of refinement-based verification, where a verification of system properties can be treated as a validation of a system specification with respect to the specification of the properties.

1 Motivation

Embedded systems is one of the most challenging fields of systems engineering: such a system must most of the time meet real-time requirements, is safety critical and distributed. The current practice in the industry of ensuring that a software system fulfills its requirements is testing. However, testing can only demonstrate the presence, but not the absence of errors. Using formal methods we can not only test correctness and safety, which is not enough for such kinds of interactive systems, but also prove them: verification guarantees fulfillment of the requirements. Coupling a specification framework with a verification system will reduce the lavishness and error-proneness of system specifications. A formal specification is in general more precise than a natural language one, but it can also contain mistakes or disagree with requirements. Therefore, for safety critical systems it is not enough to have detached formal specifications – for this case verified formal specifications are needed. Having a verified formal specification we can be sure that the specification conforms to its requirements and is consistent.

In this paper we present a coupling of a specification framework with a verification system. Given system and requirements specifications, represented in a formal specification framework, our method validates the refinement relation between them by translating the specifications to a Higher Order Logic and subsequent using the theorem prover Isabelle/HOL. This approach contains also schemata for automatic correctness proofs in Isabelle/HOL of syntactic interfaces for specified system components.

In order to design systems in a step-wise, modular style we use Focus [5], a framework for formal specifications and development of interactive systems. A specification scheme of Focus is inspired by specification approaches like Z (see [25]), but the Focus framework is much more powerful and expressive – it supports a variety of specification styles which describe system components by logical formulas or by diagrams and tables representing logical formulas. Focus is preferred here over other specification frameworks since it has an integrated notion of time and modeling techniques for unbounded networks (specification replications, sheaves of channels), provides a number of specification techniques for distributed systems and concepts of refinement. For example, the B-method [1] is used in many publications on fault-tolerant systems, but it has neither graphical representations nor integrated notion of time. Moreover, the approaches B-method also is slightly more low-level and more focused on the refinement to code rather than formal specification. Formal specifications of real-life systems can become very large and complex, and are as a result hard to read and

* This work was partially funded by the German Federal Ministry of Education and Technology (BMBF) in the framework of the Verisoft project under grant 01 IS C38. The responsibility for this article lies with the author.
to understand. Therefore, it is too complicated to start the specification process in some low-level frame-
work, First-Order or Higher-Order Logic etc. directly. To avoid this problem FOCUS supports a graphical
specification style based on tables and diagrams.

In our approach we chose a prover for Higher-Order Logic, because the power of First-Order Logic is not
efficient to represent in a direct way several specifications of distributed interactive systems. As the verification
system Isabelle/HOL we have chosen Isabelle/HOL [16,27], an interactive semi-automatic theorem prover
for Higher-Order Logic. The disadvantage of only semi-automated proofs is compensated by the advantage
of using Higher-Order Logic.

The whole and detailed description of the methodology is presented in [23]. In this paper we show the
application of its main contributions on the example of a verified specification of the FlexRay communication
protocol. This protocol has been developed by the FlexRay Consortium for embedded systems in vehicles.
The advantages of FlexRay over a CAN protocol (Controller Area Network), which is the most currently
used protocol for such kind of systems, are deterministic real-time message transmission, fault tolerance,
integrated functionality for clock synchronization and higher bandwidth. The FOCUS specification [11] of
FlexRay was chosen for the case study because the protocol is very well suited for our method – its domain
is safety-critical real-time applications.

Outline. The rest of the paper is structured as follows: In Section 2 we introduce FOCUS and the represen-
tation of its main concepts in Isabelle/HOL. In Section 3 we describe the application of represented ideas
within a case study – a verified specification of the FlexRay. In Section 4 we present the related work. Finally,
in Section 5 we summarize the presented work.

2 FOCUS on Isabelle

Isabelle [16] is a specification and verification system implemented in the functional programming language
ML. Isabelle/HOL is the specialization of Isabelle for Higher Order Logic. To specify a system with Is-
abelle means creating theories. A theory is a named collection of types, functions (constants), and theorems
(lemmas). The base types in Isabelle/HOL are bool, the type of truth values and nat, the type of natural
numbers. The base type constructors are list, the type of lists, and set, the type of sets. Function types
are denoted by \( \Rightarrow \). The type variables are denoted by 'a, 'b etc. Terms in Isabelle/HOL are formed as in
functional programming by applying functions to arguments. Terms may also contain \( \lambda \)-abstractions. For a
detailed description of Isabelle/HOL see [16,27].

FOCUS is a framework for formal specifications and development of distributed interactive systems. A
distributed system in FOCUS is represented by its components\(^1\). Components that are connected by com-
munication lines called channels, can interact or work independently of each other. The channels in FOCUS
are asynchronous communication links without delays. They are directed, reliable, and order preserving. Via
these channels components exchange information in terms of messages of specified types. Messages are passed
along the channels one after the other and delivered in exactly the same order in which they were sent.

In FOCUS any specification characterizes the relation between the communication histories for the external
input and output channels. The formal meaning of a specification is exactly this external input/output
relation. The specifications can be structured into a number of formulas each characterizing a different
kind of property, the most prominent classes of them are safety and liveness properties. A specification
can be elementary or composite. Composite specifications are built hierarchically from the elementary ones.
Elementary specifications are divided into untimed, timed, and time-synchronous according to their level of
time abstraction.

A mapping of operators in FOCUS to the corresponding definitions in HOL alone is not sufficient for
the method to become easy. Because of this, we also need a specification and proof methodology. The main
point in our methodology is an alignment on the future proofs to make them simpler and appropriate for
application not only in theory but also in practice. For this we have performed a number of case studies,

\(^1\) A component in FOCUS means a “logical component” and not a physical one.
whose results have helped us to find out different problem points (like representation of mutually recursive functions, specification replications, sheaves of channels, a large number of refinement layers, etc.) and corresponding solutions for the coupling Focus and Isabelle/HOL. The proofs of some system properties can take considerable (human) time since the Isabelle/HOL is not fully automated. But considering the framework “Focus on Isabelle”, which is presented here, we can influence on the complexity of proofs already doing the specification of systems and their properties, e.g. modifying (reformulating) specification to simplify the Isabelle/HOL proofs for a translated Focus specification. Thus, the specification and verification/validation methodologies are treated as a single, joined, methodology with the main focus on the specification part.

2.1 Concept of Streams

The central concept in Focus are streams, that represent communication histories of directed channels. Streams in Focus are functions mapping the indexes in their domains to their messages. For any set of messages $M$, $M^\omega$ denotes the set of all streams, $M^\infty$ and $M^*$ denote the sets of all infinite and all finite streams respectively. $M^\leq$ denotes the set of all timed streams, $M^{\leq_\omega}$ and $M^{\leq}$ denote the sets of all infinite and all finite timed streams respectively.

A timed stream is represented by a sequence of messages and time ticks, the messages are also listed in their order of transmission. The ticks model a discrete notion of time. Specifying embedded real-time systems we always need to argue about time. The notion of time takes center stage for this kind of systems and abstracting from time we may lose very important properties, e.g. the causality property, that are not only very important for the system, but also help us to make proofs easier. Thus, the timed domain is the most important one for representation of distributed systems with real-time requirements. Therefore, the better way to represent a real-time system in Focus is to use timed specification. All input, output and local streams in such specifications are timed or time-synchronous and, by the Focus definition of timed stream, infinite.

Specification of a real-time system in the untimed frame may be in some cases shorter or more elegant from mathematical point of view, but case studies have shown, that to understand such specifications and to argue about their properties is in many cases much more difficult in comparison to the corresponding specifications in the timed frame that use causality property explicitly. Moreover, abstraction from timing aspects can easily lead to specification mistakes because of difficulties of correct abstraction.

Hence, we can restrict the Focus specification domain for representation embedded real-time systems to only timed and time-synchronous systems. This simplifies the translation into Isabelle and also allows us to concentrate on the timing properties to have not only more clear and readable specifications, but also simpler proofs about them. Considering causality (weak or strong) it is simpler and also more readable to argue not about single messages in a timed stream, but about a sequence of messages that are present in this stream at some time interval. This sequence can be in general empty, contain a single message or a number of messages. In the case of time-synchronous stream this sequence must always contain exactly one message.

The definition in Isabelle/HOL of the Focus stream types is given below. Another ways of streams formalizations are discussed in Section 4.

- **Finite timed streams** of type ‘a are represented by the type ‘a fstream, which is an abbreviation for the type ‘a list list. This type will be used in Focus specifications of real-time systems to argue about a timed stream that was truncated at some point of time.

- **Finite untimed streams** of type ‘a are represented by the list type: ‘a list. This type will be used to argue about a sequence of messages that are transmitted during a time unit.

- **Infinite timed streams** of type ‘a are represented by the type ‘a istream that represents the functional type nat ⇒ ‘a list.

- To cover all types of Focus streams the type infinite untimed streams is also specified: nat ⇒ ‘a. We do not advise to use this kind of streams to specify real-time systems, because a specification a real-time system to untimed domain implies the loss of a most important information about a system.

For easier argumentation about the behavior of a component at some time interval we have introduced a special kind of Focus tables and a number of new operators. Here only small part of them will be used,
The operator \( \text{ti}(s, n) \) yields the list of messages that occur in the timed stream \( s \) at the \( n \)th time unit, the operator \( \text{msg}_n(s) \) holds for a timed stream \( s \), if this stream contains at every time unit at most \( n \) messages. According to our representation of the timed Focus streams the operator \( \text{ti}(s, n) \) corresponds in Isabelle/HOL simply to \( s \cdot n \), to represent the \( \text{msg}_n(s) \) operator we specify in Isabelle/HOL a predicate \( \text{maxmsg\_n\_s} \) that is equal to the Focus operator modulo syntax. The whole translation schema – from Focus to Isabelle/HOL – is presented in [23].

### 2.2 Sheaves of Channels

A specified system can contain a number of copies of channel of the same type or several instances of the same component. If this number of copies is finite, fixed and small enough, we can use the simple composition kinds, but if the number of copies must be specified as some variable of type \( \mathbb{N} \) or if the number of copies is finite and fixed, but too large to have a readable system specification, the notions of sheaf of channels and replication of specifications must be used (see [5]). A sheaf of channels in Focus can be understood as an indexed set of channels.

We define a sheaf of channels \( x_1, \ldots, x_n \) as a correct one, if all the channels \( x_1, \ldots, x_n \) are of the same type and the number \( n \) is greater than zero. To represent a sheaf of timed infinite streams \( x_1, \ldots, x_n \) of some type \( \text{Streamtype} \) in Isabelle/HOL we propose to use the following kind of functional types:

\[
\text{types nStreamtype = "nat ⇒ streamtype istream"}
\]

The corresponding bounds of sheaves used to specify a component \( C \) will be added as extra-parameters to the Isabelle/HOL predicates which represent the semantics of the component \( C \) (see Sections 3.2 and 3.3 for examples).

A sheaf will be specified in Isabelle/HOL as a single variable of corresponding type, e.g. the sheaf \( x_1, \ldots, x_n \) will be represented as a variable \( nX \) of type \( nStreamtype \). To translate the Focus formula over channels (streams) from a sheaf, e.g. to say that the predicate \( p \) is true for any stream of the sheaf \( \text{send}_1, \ldots, \text{send}_n \) (in Focus this formula is represented by \( \forall i \in [1..n] : p(s_i) \)) the following notation can be used²: \( \forall i < n. p \text{(nSend i)} \).

To argue about sheaves of channels in Isabelle/HOL we need to make sure that the sheaf is nonempty³. For this propose the Isabelle/HOL predicate \( \text{CorrectSheaf\_n} \) is used. This predicate is true, if the number \( n \) of channels is greater than zero.

### 2.3 Specifications and the Concept of Refinement

Focus specifications can be elementary or composite. Syntax of an elementary specification looks like follows:

\[
\begin{array}{c}
\text{Name (Parameter\_Declarations)} \quad \text{Frame\_Labels} \quad \text{Input\_Declarations} \quad \text{Output\_Declarations} \\
\text{in} \quad \text{Body} \\
\end{array}
\]

\( Name \) is the name of the specification; \( Frame\_Labels \) lists a number of frame labels, e.g. \( \text{untimed}, \text{timed} \) or \( \text{time-synchronous} \), that correspond to the stream types in the specification (see Section 2.1); \( Parameter\_Declarations \) lists a number of parameters (optional); \( Input\_Declarations \) and \( Output\_Declarations \) list the declarations.

² The relation \( < \) must be used, because the elements in Isabelle/HOL are counted from 0, in contrast to Focus, where the count goes from 1.

³ In Focus this is automatically true: the notation \( x_1, \ldots, x_n \) implies that \( 0 < n \).
of input and output channels respectively. Body characterizes the relation between the input and output streams, and can be a number of formulas, or a table, or diagram or a combination of them.

**Definition 1.** For any timed elementary specification $S$ we define its semantics, written $[S]$, to be the formula:

$$ i_s \in I_S^\infty \land o_s \in O_S^\infty \land B_S $$ (1)

where $i_S$ and $o_S$ denote lists of input and output channel identifiers, $I_S$ and $O_S$ denote their corresponding types, and $B_S$ is a formula in predicate logic that describes the Body.

We define semantics of an elementary specification in Isabelle/HOL in the same way as it is defined in Focus: as a predicate that describes the relation between the input and output stream (the Body).

**Composite specifications** are built hierarchically from elementary ones using constructors for composition and network description and can be represented in the graphical, the constraint and operator style. Semantics of a composite Focus specification is defined in [5] as follows:

**Definition 2.** For any composite specification $S$ consisting of $n$ subspecifications $S_1, \ldots, S_n$, we define its semantics, written $[S]$, to be the formula:

$$ [S] \text{ def } = \exists l_S \in \text{istream}_S : \bigwedge_{j=1}^{n} [S_j] $$ (2)

where $l_S$ denotes a list of local channel identifiers and $L_S$ denotes their corresponding types.

We define semantics of a composite specification in Isabelle/HOL analogous: a composite specification $S$ is a predicate

$$ \exists l_S \in \text{istream}_S : \bigwedge_{j=1}^{n} \text{predicate}_{S_j} $$ (3)

where $l_S$ denotes a list of local channel identifiers and $\text{istream}_S$ denotes their corresponding types, and $\text{predicate}_{S_j}$ denotes the predicate is a representation in Isabelle/HOL of the Focus specification $S_j$.

The approach “Focus on Isabelle” contains a number of Isabelle/HOL theories and the corresponding schemata to prove correctness of the relations between the sets of input, output and local channels of a specified system. E.g., the following properties must be proven for every composite specification: No input stream $i$ of a system $S$ can be an output stream of any subcomponent.

$$ i_S = \bigcup_{j=1}^{n} (i_{S_j} \in I_S^\infty) \setminus l_S \land i_S \cap \bigcup_{j=1}^{n} o_{S_j} = \emptyset $$

Every local stream $l$ of the system $S$ (consisting of $n$ subcomponents) must be both an input stream of some subcomponent $S_{j_1}$, $1 \leq j_1 \leq n$, and an output stream of some subcomponent $S_{j_2}$, $1 \leq j_2 \leq n$ ($j_1 \neq j_2$):

$$ l_S = \bigcup_{j=1}^{n} i_{S_j} \cap \bigcup_{j=1}^{n} o_{S_j} $$

The proof schemata (for the correctness properties) specified in our approach are standard and can be used automatically for all refinement layers. If the proof fails, the specification of the corresponding set is incorrect and must be changed. But the main part of proofs about a system that we need are the proofs that a system fulfills its requirements.

In Focus we can have a general specification $S_0$ of a system that corresponds to the formalization of system requirements. To show that a concrete specification $S_n$, which we get after $n$ refinement steps, fulfills the system requirements, we only need to show that the specification $S_n$ is a refinement (see also [5] and [3]) of the specification $S_0$. For this purposes also the idea of a refinement-based verification of interactive real-time systems can be used (see [23] and [24]).
Definition 3. A specification $S_2$ is called a behavioral refinement ($S_1 \sim S_2$) of a specification $S_1$ if they have the same syntactic interface and any I/O history of $S_2$ is also an I/O history of $S_1$. Therefore, in order to show that our concrete specification $S_2$ fulfills the system requirements $S_1$, we only need to show

$$[S_2] \Rightarrow [S_1]$$

(4)

Formally, we need to show that any I/O history of $S_2$ is an I/O history of $S_1$, but $S_1$ may have additional I/O histories. In Isabelle it means to prove that the formula that corresponds to $[S_2]$ implies the formula that corresponds to $[S_1]$. This definition of refinement does not exclude that the set of I/O histories of $S_2$ can be empty. It means $[S_2]$ is false and the refinement relation is true. This can happen if the specification $S_2$ is inconsistent. Thus, the consistency of $S_2$ must also be proved doing the proof in Isabelle/HOL.

3 Case Study: From Specification to Verification of the FlexRay Communication Protocol

The feasibility of the approach “FOCUS on Isabelle/HOL” was evaluated on a number of case studies that cover different application areas: process control, memory and processing components, data transmission etc. The proofs for these case studies have taken from 200 to ca. 2000 lines of proof. The following has been done within every case study: The FOCUS specifications of all components of the system have been translated schematically to Isabelle/HOL and the refinement relation between the requirement and the architecture specification of the system has been proved. The correctness of the input/output relations has been also proved for all components of the system (automatically, according to the specified proof schemata). The FlexRay case study was chosen for the case study to show how we can deal with sheaves of channels ans parameters, as well as with specification replications.

FlexRay [8] is a static time division multiplexing network protocol developed for embedded systems in vehicles. It is based on deterministic real-time message transmission between a number of nodes. FlexRay contains a set of complex algorithms to provide the communication services. From the view of the software layers above FlexRay only a few of these properties become visible. The most important ones are static cyclic communication schedules and system-wide synchronous clocks. These provide a suitable platform for distributed control algorithms as used e.g. in drive-by-wire applications. The formalization described here is based on the “Protocol Specification 2.0”[8]. A formal verification of the clock synchronization algorithm and of the bus guardian of FlexRay is in progress at INRIA [28].

The static message transmission model of FlexRay is based on rounds. FlexRay rounds consist of a constant number of time slices of the same length, so called slots. A node can broadcast its messages to other nodes at statically defined slots. At most one node can do it during any slot.

We have presented the first version of the formal specification of FlexRay in FOCUS in [11]. We have discussed the general introduction to the FlexRay formalization also in [12] and [13]. Now we are going to present a schematically translation of this formal specification into Isabelle/HOL using the representation of FOCUS streams presented above. After that the proof of the refinement lemma is discussed – the refinement lemma says that the FlexRay architecture specification fulfills the FlexRay requirements. Since the overall representation of FlexRay in FOCUS and Isabelle/HOL as well as the proofs of auxiliary lemmas are too extensive for this paper, we describe here only some aspect of the specifications and proofs, and show only a simple and short parts of the specifications to give a feeling how the approach works. For the technical details of the case study we would like to refer to [23].

3.1 Representation of Datatypes

The specifications of the following types are equal modulo syntax to the corresponding types in the FOCUS specification of FlexRay. The type Message consists of a slot identifier slot and the payload data. The type of payload is defined in FOCUS as a finite list of type FT_CNI_Entity that consists of a message ID of N (type of natural numbers) and data of type DataType. Because the type DataType is not specified in
Focus exactly (to have a polymorphic type), it must be polymorphic in Isabelle/HOL also. This implies furthermore that the types \texttt{FT\_CNI\_Entity} and \texttt{Message} are polymorphic. The type \texttt{Config} represents the bus configuration and contains the scheduling table \texttt{schedule} of a node and the length of the communication round \texttt{cycleLength}. A scheduling table of a node consists of a number of slots in which this node should be sending a message with the corresponding identifier (identifier that is equal to the slot). We present here as example only the Isabelle/HOL representation of the type \texttt{Message}:

\begin{verbatim}
record 'a Message =
  slot :: nat
  data :: "('a FT\_CNI\_Entity) list"
\end{verbatim}

The types \texttt{nMessage} and \texttt{nSlot} are used to represent sheaves of channels of corresponding types. In a similar way we define the type \texttt{nConfig} for the list of parameter constants \(c_1, \ldots, c_n\) of the type \texttt{Config}.

\begin{verbatim}
types 'a nMessage = "nat ⇒ ('a Message) istream"
types nSlot = "nat ⇒ nat istream"
types nConfig = "nat ⇒ Config"
\end{verbatim}

### 3.2 Requirements Specification

The requirements specification FlexRay, which is represented in FOCUS, contains the assumptions (\texttt{asm}) and guarantees (\texttt{gar}) for the FlexRay network (an A/G specification). This means whenever input from the environment behaves in accordance with the assumption, the specified component is required to fulfill the guarantee. The assumptions are the following ones:

- For all nodes the scheduling tables of the system are disjoint.
- The communication cycles have the same length on each node.
- In every time interval on each input channel \texttt{return\_i} can come at most one FlexRay frame (a message of type \texttt{Message}).

Having this assumptions the specification requires fulfillment of the following properties:

- The message transmission is correct: If at some slot a node should be sending a message according to it’s scheduling table, this message is requested from the local buffer and is sent over the channels to the other nodes of the system.
- In every time interval on each output channel \texttt{get\_i} as well as can come on each output channel \texttt{store\_i} at most one FlexRay frame.

\begin{verbatim}
FlexRay (const c1, ..., cn ∈ Config) timed
  in return1, ..., returnn : Message
  out store1, ..., storn : Message; get1, ..., getn : Slot
  \text{asm } ∀ i ∈ [1..n] : msg_i(return_i)
  \text{DisjointSchedules}(c1, ..., cn)
  \text{IdentcCycleLength}(c1, ..., cn)

  \text{gar } MessageTransmission(return1, ..., returnn, store1, ..., storn, get1, ..., getn, c1, ..., cn)
  \forall i ∈ [1..n] : msg_i(get_i) \land msg_i(store_i)
\end{verbatim}
The Focus predicate *MessageTransmission* defines the correct message transmission according the definition above: If at time $t$ the node $k$ should be sending a message according to its scheduling table, this message is requested over the channel $get_k$ and received over the channel $return_k$ from the local buffer. This message is then sent over the channels $store_j$, $j \in [1..n]$, $j \neq k$ to the other nodes of the system.

The predicates *DisjointSchedules*, *IdenticCycleLength*, *MessageTransmission* from the Focus specifications are equal modulo syntax to predicates the same name that we specify in Isabelle/HOL. The predicate *DisjointSchedules* is defined to be true for a sheaf of channels, if all bus configurations have disjoint scheduling tables. The predicate *IdenticCycleLength* is defined to be true, if all bus configurations have the equal length of the communication round.

The predicate *FlexRay* represents the semantics of the Focus specification *FlexRay*. In the case the relation between the input and output streams is specified in Focus as a number of formulas the corresponding representation of the semantics will be the conjunction of these formulas. The first argument of the predicate *FlexRay* corresponds to the number $n$ of streams in the sheaves $store_1, ..., store_n$, $get_1, ..., get_n$, $return_1, ..., return_n$ and the number of parameters $c_1, ..., c_n$.

constdefs

FlexRay ::

"nat ⇒ 'a nMessage ⇒ nConfig ⇒ 'a nMessage ⇒ nSlot ⇒ bool"

"FlexRay n nReturn nC nStore nGet
≡ (CorrectSheaf n ∧ (∀ i < n. maxmsg 1 (nReturn i)) ∧ (DisjointSchedules n nC) ∧ (IdenticCycleLength n nC)
→ (MessageTransmission n nReturn nStore nGet nC) ∧ (∀ i < n. maxmsg 1 (nGet i) ∧ maxmsg 1 (nStore i))))"

### 3.3 Architecture Specification

The architecture of the FlexRay communication protocol is specified as the the Focus specification *FlexRayArch* that is an assumption/guarantee one. The assumption part of the specification *FlexRayArch* is the same as of the specification *FlexRay*. The guarantee part is represented by the specification *FlexRayArchitecture* (see below) that is a composite one and consists of the component *Cable* and $n$ components *FlexRay_Controller* (for $n$ nodes). The specification *FlexRayArch* is a refinement of the specification *FlexRay* – this will be shown in Section 3.4.

```
FlexRayArch (const c1, ..., cn ∈ Config) tim
in return1, ..., returnn : Message
out store1, ..., storen : Message; get1, ..., getn : Slot

asm ∀ i ∈ [1..n] : msg1 (returni)
DisjointSchedules(c1, ..., cn)
IdenticCycleLength(c1, ..., cn)

gar FlexRayArchitecture (const c1, ..., cn ∈ Config)
(return1, ..., returnn, store1, ..., storen, get1, ..., getn)
```

The component *FlexRay-Controller* is a composite one and consists of the components *Scheduler* (has a timing control function: decides if the node is allowed to send in the current slot) and *BusInterface* (represents the receive and the send of messages). The component *Cable* describes the transfer properties: in every
time unit only one of the streams has any messages to transfer, if some node sends a message, this message will be received by all nodes, etc. The predicate \textit{FlexRayArchitecture} represents here the semantics of the corresponding FOCUS specification. Please note, that \textit{nSend} and \textit{recv} are the local channels in the composite component.

\begin{center}
\begin{tikzpicture}

\node at (-2,0) {FlexRayArchitecture \texttt{(const }c_1, ... , c_n \in \text{Config})};

\node at (-2,-2) {glass-box =};

\node at (-2,-4) {flexray-Controller};

\node at (-2,-6) {cable};

\node at (-4,-8) {store1 : \text{Message}};

\node at (-4,-9) {FlexRay-Controller \{cn\}};

\node at (-4,-10) {get1 : \text{Slot}};

\node at (-4,-11) {return1 : \text{Message}};

\node at (-4,-12) {send1 : \text{Message}};

\node at (-4,-13) {recv : \text{Message}};

\node at (-4,-14) {storen : \text{Message}};

\node at (-4,-15) {getn : \text{Slot}};

\node at (-4,-16) {returnn : \text{Message}};

\node at (-4,-17) {sendn : \text{Message}};

\end{tikzpicture}
\end{center}

\textbf{constdefs}

\textit{FlexRayArchitecture} ::

\texttt{"nat \Rightarrow 'a nMessage \Rightarrow nConfig \Rightarrow 'a nMessage \Rightarrow nSlot \Rightarrow bool"}

\texttt{"FlexRayArchitecture \ n \ nReturn \ nC \ nStore \ nGet}

\texttt{≡}

\texttt{∃ \ nSend \ recv.}

\texttt{CorrectSheaf \ n \ (Cable \ n \ nSend \ recv) \ ∧}

\texttt{(∀ \ i < n. \ FlexRay_Controller \ (nReturn \ i) \ \ recv \ (nC \ i)}

\texttt{\ (nStore \ i) \ (nGet \ i) \ \ (nSend \ i))"}

\subsection{Proof of the Refinement in Isabelle/HOL}

The lemma \textit{main_fr_refinement} says that the specification \textit{FlexRayArch} is refinement of the specification \textit{FlexRay}: the predicate \textit{FlexRayArch} that represents the semantics of the architecture specification \textit{FlexRayArch} implies the \textit{FlexRay} that represents the semantics of the requirements specification \textit{FlexRay}:

\textbf{lemma} \textit{main_fr_refinement}:

\texttt{∧ \ n \ nReturn \ nC \ nStore \ nGet.}

\texttt{FlexRayArch \ n \ nReturn \ nC \ nStore \ nGet \ \Longrightarrow \ FlexRay \ n \ nReturn \ nC \ nStore \ nGet}

To prove this lemma we used the definitions of the predicates, used in the specification, Isabelle/HOL reasoning methods \texttt{clarify}, \texttt{clarsimp} and \texttt{auto} that perform rewriting and classical reasoning automatically, rules of natural deduction etc. To prove the resulting subgoals in more structural way, we have proven a number of auxiliary lemmas \cite{23}. The proof of this lemma takes only ca. 300 lines, but rather in a specification of such an observable size we have found out a number of inconsistencies (see the next subsection).

\subsection{Results of the Case Study}

In this case study we have shown how we can deal with sheaves of channels and parameters, as well as with specification replications. The FOCUS specifications of all components of the FlexRay system were translated
schematically to Isabelle/HOL and the refinement relation between the requirement and the architecture specification of the system was proved. The correctness of the input/output relations was also proved for all components of the system.

Doing the verification in Isabelle/HOL of the first versions of the FOCUS specification of FlexRay we found out a number of inconsistencies, which were corrected in the final version: lost assumptions, too weak properties of the subcomponents, etc. For example, in the requirement specification FlexRay the assumption, that in every time interval on each input channel return can come at most one FlexRay frame, was loosed. The resulting formal specification of FlexRay is a verified specification that guarantees fulfillment of the requirements.

4 Related Work

The first attempt to represent a simplified version of the FOCUS syntax [4] (without representation of time, modeling techniques for unbounded networks, etc.) in a verification system was done by B. Schätz and K. Spies [20]. In this approach the HOLCF specialization of the theorem prover Isabelle was chosen. HOLCF (see [19] and [15]) is the definitional extension of Church’s Higher-Order Logic with Scott’s Logic for Computable Functions that has been implemented in Isabelle. HOLCF supports standard domain theory but also coinductive arguments about lazy datatypes. The main disadvantage of using HOLCF in practice is difficulty of logic understanding in comparison to HOL.

The first attempt of coupling of FOCUS with an automatic verification system was done by J. Schumann and M. Breitling [21]. As the verification system was chosen SETHEO [14], an automatic theorem prover for proving the unsatisfiability of formulas in First-Order Clause Logic. This case study of J. Schumann and M. Breitling has shown that such a coupling is in principle possible, but there are also a number of problems and open questions. In our approach we chose a prover for Higher-Order Logic, because the power of First-Order Logic is not enough to represent in a direct way several specifications of distributed interactive systems.

The central concept in FOCUS are streams, and there are different ways to formalize them. They have different advantages and disadvantages. One of the ways to represent FOCUS streams is to use the coalgebraic approach [10], but the representation of FOCUS streams in a coalgebraic domain [22] is more difficult to understand in practice as an inductive one in Isabelle/HOL – for the case of restriction the specification domain to only real-time systems.

The representation of FOCUS streams in Isabelle/HOLCF that was done by D. von Oheimb [26] does not cover representation of FOCUS timed streams, which are the most important for the specification of real-time systems. The further development of the FOCUS stream representation in Isabelle/HOLCF is presented by the approach of B. Gajanovic and B. Rumpe [9], that covers HOLCF specification of many important operators on streams, like concatenation, delete prefixes, take an element of the stream etc., as well as the properties of these operators. But this representation of the FOCUS streams in Isabelle/HOLCF covers only the general representation of streams, and abstracts from the representation of timing aspects as well as from the question how to deal with proofs for such translated specifications. Thus, the representation of the timing aspects can be done as an extension, but the resulting construction will be much more complicated than is needed for system specification in the timed domain.

To represent FOCUS streams in Isabelle/HOL we need to take into account both properties of FOCUS and Isabelle/HOL. In Isabelle/HOL we can represent streams in the two following ways. The first way is the representation of streams as

\[ \alpha \text{ seq} = \mathbb{N} \to \alpha \text{ option} \]

where the datatype

\[ \alpha \text{ option} \equiv \text{None} \mid \text{Some } \alpha \]

and \text{None} denotes a non-existing element (see [18], [17]). This approach is claimed in [7] to be inconvenient in practice to prove equalities of arbitrary functions, because for every operation over such a stream the notion
of stream normal form must be used explicitly to avoid the case in which None appears within a sequence the specification, but it is not straightforward to construct the normal form.

The second way, the representation of streams as the disjoint sum of finite stream (lists) and infinite streams (functions), has been chosen by C.-T. Chou and D. Peled [6] and by S. Agerholm [2]. The main difficulties (see also [7]) in using this approach arise from type comparison in the definitions of stream processing functions – all inputs are finite, all inputs are infinite or some of them are finite and some infinite – several versions of function definitions are needed. But in the case we work only with timed frames we do not have this disadvantage, because we deal with timed streams that are always infinite. Moreover, such a representation in this case leads to more clear specification structure. This representation is the most natural one to FOCUS and was in our approach for the our representation of FOCUS in Isabelle/HOL.

5 Conclusion

A formal specification is more precise than a natural language one, but it can also contain mistakes or disagree with requirements. Therefore, for safety critical systems it is not enough to have detached formal specifications – in this case formal verification is needed. This is the only way to be sure that the specification conforms to its requirements and is consistent. In this paper we have introduced the coupling of the formal specification framework Focus in the generic theorem prover Isabelle/HOL. The result of the coupling of the formal specification framework Focus in the generic theorem prover Isabelle/HOL is the framework “Focus on Isabelle”. Given both specifications represented in Focus, we prove using the theorem prover Isabelle/HOL that the system specification is a refinement of requirements specification of the system, i.e. that this specification fulfills its requirements. Using the framework we also can make automatic correctness proofs in Isabelle/HOL of the syntactic interfaces for specified system components.

The presented case study, verification of the FlexRay specification, showed the feasibility of the approach. The properties of the protocol were formalized as the requirements specification, and its architecture was formalized as the corresponding specification. The system specification was subsequently verified according to the FlexRay requirements – the refinement relation between them is proved. Doing the verification in Isabelle/HOL of the first versions of the Focus specification of FlexRay we found out a number of inconsistencies like loosed assumptions, which were corrected in the final version of the specification. The resulting specification is a verified one.

References


The timed streams must be infinite because time never halts.
A Separation Logic Framework in HOL

Thomas Tuerk
University of Cambridge Computer Laboratory
William Gates Building, JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom
http://www.cl.cam.ac.uk

Abstract. Separation Logic has proved to be a powerful technique for program verification. There are tools based on ideas of separation logic and even some implementations in theorem provers. However, these implementations all concentrate on the verification of programs written in one specific language. In contrast, I propose to build a framework inside the HOL theorem prover that concentrates on separation logic itself. This framework should be general enough to express different flavours of separation logic and it should be easily instantiable for different programming languages.

In this paper, a formalisation of Abstract Separation Logic in HOL is presented as a first step towards implementing the proposed framework. This presentation contains an initial case study – an implementation of a tool similar to Smallfoot – which gives some insights into the suitability of abstract separation logic for this purpose.

1 Introduction

Separation logic is an extension of Hoare logic that allows local reasoning about mutable data structures, especially structures using pointers. It’s been introduced by O’Hearn, Reynolds and Yang in 2001 [7,9] based on earlier ideas by Burstall, Reynolds, Ishtiaq and O’Hearn. The main idea of separation logic is the usage of a spatial conjunction $p \ast q$ that asserts that the formulas $p$ and $q$ hold on separate parts of the state. This notion of separate parts allows local reasoning and an elegant solution to aliasing problems. Moreover, it enables separation logic to be extended to concurrent programs in a natural way [3].

1.1 Introductory Example

Classically, separation logic uses states consisting of a stack and a heap. Consider such states and, as an example, an update of the heap at the location stored in a stack-variable $x$ with the value 2 ($[x] := 2$). In order to reason about this assignment, separation logic needs the precondition, that $x$ is allocated ($x \mapsto \_$. Using this precondition, one can conclude that after executing the assignment $x$ points to the value 2 ($x \mapsto 2$). This whole reasoning is captured by the validity of the following Hoare triple $\{ x \mapsto \_ \} [x] := 2 \{ x \mapsto 2 \}$. In contrast to classical Hoare logic, the pre- and postcondition have to mention at least, that $x$ is allocated. $\{ x \mapsto \_ \} [x] := 2 \{ \text{emp} \}$ for example does not hold. So, one cannot easily remove parts of the postcondition.

On the other hand, the conditions can be safely extended by an arbitrary context using the spatial conjunction operator. This is due to the separation that this operator provides and locality properties of programming languages. This property gives rise to the following inference rules for extending the conditions with a frame and parallel composition, which capture the essence of local reasoning:

\[
\frac{P \text{ prog } Q}{P \ast R \text{ prog } Q \ast R} \quad \frac{P_1 \text{ prog } Q_1 \quad P_2 \text{ prog } Q_2}{(P_1 \ast P_2) \text{ prog } (Q_1 \ast Q_2)}
\]

1.2 Existing Implementations of Separation Logic

Due to local reasoning, separation logic scales much better than classical Hoare logic to the verification of large programs. Moreover, even for simple, small pieces of programs separation logic proofs are often much more succinct and easier to read. Separation logic has become more and more popular during the last few
years. There are several implementations: Smallfoot [2], SLayer and SpaceInvader are probably some of the best known examples. There are also formalisations inside theorem provers. There are several implementations of separation logic in Isabelle/HOL [10,11] and one in Coq [6], which is used and extended by the Concurrent C-Minor Project [1]. There is also related work by Andrew Ireland et al [5], but I learned about it too late to discuss it here.

1.3 Motivation for the Proposed Framework

The implementations mentioned above all focus on one specific programming language (mostly some C-like imperative language) and take design decisions with respect to this language. While in most cases a specific language is all you are interested in and while it allows better automation and perhaps simplified and more intuitive formalisations for this specific language, it makes reusing of these formalisations very difficult and distracts from the core features of separation logic.

The differences and problems with reuse start with simple design decisions like whether the heap is modelled as a function from integers to integers or whether it can contain arrays of integers. Is it perhaps even more appropriate to explicitly model the finiteness of memory by having 32- or 64-bit words instead of integers? Problems increase if you consider changes to the logic itself like equipping values on the stack with explicit read/write permissions [8]. Even worse, one may not use the classical stack/heap model at all and be interested in verifying assembler code that operates on a fixed set of registers and a chunk of memory instead of a heap and a stack. In short, there are a lot of different flavours of separation logic, which share a large common part, but may dramatically vary in detail.

Therefore, I suggest building a framework for separation logic inside the HOL theorem prover that concentrates on separation logic itself instead of a concrete programming languages.

1.4 Structure of this Paper

In the next section, I briefly state some ideas for such a separation logic framework in HOL. Then, a first step towards implementing such a framework is presented: a formalisation of Abstract Separation Logic in HOL. The paper will conclude with first experiences of instantiating abstract separation logic and plans for future work.

2 Ideas for a Separation Logic Framework

I suggest building a framework for separation logic inside the HOL theorem prover. The main part of this framework should be general, but it should provide support for instantiating it for different concrete languages and applications. I hope that these instantiations can be significantly different while still keeping a large common part. As case studies one could try to implement:

- a completely automated tool – similar to Smallfoot [2] – for a simple imperative language (this case study is mainly done);
- a tool to interactively verify programs written in a more complicated imperative language;
- an interactive separation logic calculator that may be used to formalise long, complicated hand-proofs found in separation logic papers;
- a tool to verify assembler programs.

I hope that splitting the formalisation into a general part and instantiations will add to clarity and help to keep a lot of proofs simple. For example, one should be able to neglect the complicated structure of states some separation logics use (e.g. [8]) and mainly work with abstractions. However, choosing these abstractions has a huge impact and is far from obvious. On the one hand, the abstractions should be abstract enough to model a large variety of separation logics and to keep the general part as simple as possible. On the other hand, they need to be concrete enough to be able to convince people that the formalisation of a concrete programming language in the framework is reasonable or even intuitive. I believe, that Abstract Separation Logic [4] is a good starting point to build such a framework.
3 Abstract Separation Logic

Abstract Separation Logic was introduced by Calcagno, O’Hearn and Yang [4]. While most separation logics operate on states consisting of a stack and a heap, abstract separation logic can use arbitrary states. A partial function \( \bullet \) is used to combine these states. Two states \( s_1 \) and \( s_2 \) are separate, iff \( s_1 \bullet s_2 \) is defined. Using this notion, one can easily define the spatial conjunction operator \( * \) as follows:

\[
P * Q := \{ s | \exists p, q. (p \bullet q = s) \land p \in P \land q \in Q \}
\]

Intuitively, this means, that a state \( s \) satisfies \( P * Q \) iff it can be split into two separate states \( p \) and \( q \) such that \( p \) satisfies \( P \) and \( q \) satisfies \( Q \). Other standard separation logic constructs can be defined in a natural way as well. However, in order for these definitions to be useful, the combination function \( \bullet \) has to satisfy some properties: a neutral element \( u \) has to exist, such that the set of states \( \Sigma \) forms a separation algebra with \( \bullet \) and \( u \), i.e. \( (\Sigma, \bullet, u) \) is a cancellative, partial commutative monoid.

3.1 Programming Language

The programming language used by abstract separation logic is abstract as well. An action \( \text{act} \) is a function from a state \( s \) to a set of states \( S \) or a special failure state \( \top \). If \( \text{act}(s) = \top \), then an error may occur during the execution of the action starting in state \( s \). This is used to model for example that an action might try to dereference a null-pointer or read an unallocated location on the heap. If \( \text{act}(s) = S \), then no error will occur and after executing the state will be one of the states in \( S \). If \( S \) is the empty set, the action does not terminate. Thus, \( \text{act}(s) = S \) can be used to express nondeterminism and nontermination.

Based on this notion of actions, a Hoare triple \( \{ P \} \text{act} \{ Q \} \) holds, iff for all states \( p \) that satisfy the precondition \( P \) the action does not fail, i.e. there is a set of states \( S \) with \( \text{act}(p) = S \), and leads to a state that satisfies the postcondition \( Q \), i.e. \( S \subseteq Q \). Notice, that this describes partial correctness, since a Hoare triple is trivially satisfied, if \( \text{act} \) does not terminate, i.e. if \( S \) is empty.

As explained above, it is an essential feature of separation logic, that a specification can be safely extended by an arbitrary frame \( R \). This is the essence of local reasoning. Therefore, abstract separation logic only uses actions that satisfy this property. These actions are called local. More explicitly, an action is called local, iff it satisfies the following inference rule for all \( P, Q \) and \( R \):

\[
\frac{\{ P \} \text{act} \{ Q \}}{\{ P * R \} \text{act} \{ Q * R \}}
\]

Using only local actions is not a big restriction, since most programming languages just use local actions anyhow.

Abstract separation logic provides some operations to extend the set of local actions provided by the user to a programming language. This extension guarantees that all programs written in it are local actions themselves. There are e.g. a sequential composition operator \( (\cdot) \), a nondeterministic choice operator \( (+) \) and a Kleene star operator \( (\ast) \). One important predefined local action is \( \text{assume} \) for a predicate \( B \). The predicate \( B \) has to be intuitionistic and its intuitionistic negation is denoted by \( \neg_iB \). Intuitionistic predicates are not discussed here for reasons of brevity. Given a state \( s \) the action \( \text{assume} \ B \)

- skips, iff \( B \) holds in all extensions of \( s \) by a frame;
- diverges, iff \( B \) does not hold in any extension of \( s \) by a frame;
- fails otherwise.

For example, in a usual setting \( \text{assume} \ x \neq 0 \) skips, iff \( x \) is defined in the state \( s \) and does not contain the value 0, diverges iff \( x \) is defined and contains the value 0 and fails, iff \( x \) is not defined. Since abstract separation logic considers partial correctness, divergence has the effect of satisfying any specification. Therefore, conditional execution and loops can be defined using nondeterminism, Kleene star and assume:

\[
\text{if } B \text{ then } \text{prog}_1 \text{ else } \text{prog}_2 := (\text{assume } B; \text{prog}_1) + (\text{assume } \neg_iB; \text{prog}_2) \\
\text{while } B \text{ do } \text{body} := (\text{assume } B; \text{body}^\ast); \text{assume } \neg_iB
\]
Notice, that the semantics of a while-loop can be seen as nondeterministically choosing a natural number \( n \) and unrolling the loop \( n \) times. If the wrong number for a particular input has been chosen, the execution is aborted by one of the assumptions. If the loop would not terminate, no such number exists and every choice is aborted by the assumptions. This captures the semantics of loops in a simple, abstract way, but it is not intuitive.

Abstract separation logic defines constructs for expressing parallel programs as well. There is a parallel composition operator \(||\) that executes two programs in parallel. Moreover, there are semaphore operations. However, for reasons of brevity these operations are not presented here. Neither is the exact semantics discussed. However, the part described here should give a glimpse of what constructs abstract separation logic provides and how its abstract programming language can be instantiated to a concrete one. For more details please refer to the original paper about abstract separation logic [4].

3.2 Inference Rules

Using the detailed semantics of abstract separation logic, one can prove high-level inference rules correct. The goal is to have sufficiently expressive inference rules to verify specification in this high-level view instead of using the detailed semantics all the time. Some important inference rules, that are valid in abstract separation logic are:

\[
\begin{align*}
\{ P \} \text{p} \{ Q \} & \quad \{ P \} \text{p} \{ Q \} \quad \{ P \} \exists \{ R \} \\
\{ P \} \text{p} \{ P \} & \quad \{ P \} \text{p} \{ Q \} \quad \{ P \} \text{p} \{ Q \} \\
\{ P \} \exists \{ P \} & \quad \{ B \land P \} \text{p} \{ Q \} \quad \{ \neg \iota \land B \land P \} \text{p} \{ Q \} \\
\{ B \land P \} \text{p} \{ P \} & \quad \{ P \} \text{while} \ B \ do \ p \{ \neg \iota \land P \}
\end{align*}
\]

3.3 HOL implementation

I formalised abstract separation logic as described in the paper by Calcagno, O’Hearn and Yang [4]. This formalisation includes the specification logic, concurrent semantics, proofs for all inference rules and most lemmata found in this paper. There are also some extensions. The HOL sources can be found in the HOL repository.

The formalisation consists of a mixture of deep and shallow embeddings. In general, I tried to keep it as flexible as possible for instantiations and used shallow embeddings. However, the programs are defined by a deep-embedding. This embedding depends on several free type variables used to instantiate elementary actions, predicates etc. To instantiate the formalisation of abstract separation logic, one has to provide among other things a concrete type for states, a partial function \( \cdot \) and a neutral element \( u \) for combining states, concrete types for elementary actions and predicates and functions assigning a semantic to these actions and predicates. All these things needed for an instantiation are collected in an environment term. One has to prove that this environment is valid, which includes for example that \( \cdot \) forms a separation algebra or that all used elementary actions are local ones. A lot of functions defined in the formalisation take this environment as an argument and theorems use the validity of the environment as a precondition.

I also formalised some extensions. The largest extension are procedures. Their semantics are defined by replacing a function call with the body of the function. The formalisation can handle mutually recursive

1 The HOL repository can be found at http://hol.sourceforge.net. The formalisation of abstract separation logic and case studies are located in examples/separationLogic.
definitions and there are some inference rules to eliminate recursion during the verification. The usual separation logic predicates like magic wand or septraction are predefined. Additionally, there are predicates like separation logic quantifiers or a definition scheme for recursive predicates. Moreover, there are also definitions and inference rules for commonly used imperative constructs like loops or conditional execution. However, there is still a lot to add and I expect the formalisation to grow with future instantiations.

4 First Instantiations and Resulting Experiences

After formalising abstract separation logic in HOL, I tried to instantiate it to the flavour of separation logic presented in Variables as Resource in Hoare Logic by Parkinson, Bornat and Calcagno [8]. This separation logic treats stack-variables as resources, i.e. it extends the separation idea from the heap to the stack. To this end, each variable on the stack is equipped with a read/write permission.

Proving that the resulting operation to combine states consisting of these extended stacks and classical heaps forms a separation algebra took some time but is straight-forward. However, there were some problems with formalising the programming language. The main problem is locality: like most languages the language used in Variables as Resource [8] has a concept of local variables. Consider for example the following pseudo-code for traversing a list:

```plaintext
list_traverse(x) {
    local t;
    t := x;
    if (t != NULL) then {
        list_traverse(t.next_list_element);
        do something with data in t;
    } else {done}
}
```

The local variable t needs to address a new and different location on the stack in each recursive call of list_traverse. A common solution is to informally demand that a fresh variable z is chosen every time and t is replaced by z. However, there are some problems formalising this notion of freshness in an intuitive way.

Abstract separation logic does not provide a notion of locality itself. However, it contains nondeterministic choice and assumptions. These can be used to nondeterministically choose a variable that is not present in the current state. Being not present in the current state is sufficient; however, it is not straight-forward to see that it does not matter if a variable is chosen that will be used in the future by a different part of the program. This semantics for local variables is abstract, fits easily into the framework, leads to simple proofs for corresponding inference rules etc. Unfortunately, it is not easy to see that this semantics is really the one intended. It’s not intuitive.

In my opinion it is preferable to use a semantics that explicitly hides the current value when the scope of a local variable is entered and restores it afterwards. However, hiding and restoring are not local actions. Moreover, these hiding and restoring operations would need to respect threads somehow, since using a local variable t in one thread should not influence the usage of another variable t in a different thread. It would require serious modifications to allow variables to have different values depending on the thread that interprets them. Even if one is willing to apply such modifications, it is not obvious how such changes might look like. Should one just add thread identifiers and leave the details completely to the instantiations? This would keep the abstract level nice and clean, but cause a large part of the formalisations to be repeated over and over again, since probably most instantiations need a concept similar to local variables. One could try to split the current global state into one global state and local states for each thread. That is a common view and would simplify instantiations. However, the abstract part of the formalisation would become much more complicated and harder to understand.

Currently, variables denote explicit locations on the stack. So one has to manipulate either the program by replacing the variable with a different one or one has to modify the stack. Both ways are causing problems as explained above. However, one could add an extra layer of abstraction: namespaces. A namespace is a
function mapping names to values. Variables could just be names that need a namespace to be interpreted as locations on the stack. In different parts of the code or different threads the same name could point to different locations. There would be just one global state instead of the complicated structure above. Admittedly, there would be many namespaces, but their structure is very simple and operations to manipulate namespaces could be very limited. Adding namespaces would not complicate the abstract level much, but provide the needed functionality. Moreover, the concept is abstract enough to be used for other purposes as well. It could for example solve very similar problems with locally defined recursive functions.

During the discussions about the semantics of local variables, it turned out that it might be beneficial to make the model of parallel computation more suited for real world programming languages. To this end, one could switch to C-like fork/join parallelism. However, this would involve huge modifications. A rather minor change that turned out to be probably useful is to change synchronisation to conditional critical regions.

5 Current Work

Apart from these problems with locality, I did not experience major trouble with the first instantiation. However, I did not formalise the formalism described in Variables as Resource [8] exactly. The formalism described there allows locally defined mutually recursive functions. This causes problems in the current setting that could be overcome using namespaces as described above. To keep things simple at the beginning, I started formalising a mixture of the logic described in this paper with the one used by Smallfoot [2] as a case study.

Smallfoot is a completely automated tool that allows to verify specifications of programs written in a simple imperative language. This language know mutually recursive functions, but no local function definitions. Compared with other tools and implementations the separation logic used is weak. Thus, it’s a good starting example. I use the Smallfoot syntax, but a semantics that is much closer to the one used by Variables as Resource [8] than the one used by Smallfoot. In particular, the stack variables are treated as resource, i.e. they are equipped with permissions. The goal of this case study is to develop a tool, that parses Smallfoot input files and is able to verify the specifications completely automatic inside the HOL theorem prover. This goal has nearly been achieved.

The tool is already able to verify some input files completely automatic. However, some features of Smallfoot are still missing. The biggest missing feature are conditional critical regions. Support for semaphores is available at the abstract level, but there is no support for any kind of synchronisation in the Smallfoot instantiation yet. Moreover, there is no support for existantically quantified specifications. As existential quantification is used internally however, it is simple to add this feature. It would be straight-forward to add some more predicates used by Smallfoot like double linked as well. I guess that it would take a few more weeks to implement all missing features of Smallfoot.

6 Conclusion and Future Work

I already took some initial steps towards implementing a framework for separation logic inside the HOL theorem prover:

– I formalised abstract separation logic as described in the original paper [4].
– This formalisation contains extensions of abstract separation logic by
  • procedure calls,
  • standard separation logic operators,
  • common imperative programming constructs.
– I formalised a big part of the logic described in Variables as Resource [8].
– As a case study, I implemented a completely automated tool, that uses a language very similar to the one used by Smallfoot:
  • I instantiated the framework to the needed specification and programming languages.
• There are proofs of specialised inference rules.
• A parser for reading Smallfoot specifications has been implemented.
• Specialised tactics have been implemented to verify these parsed specifications.

While this completed work can just be considered as initial steps towards the proposed framework, it already took a significant amount of effort. There are about 30000 lines of proofs and about 6000 lines of ML for the automation.

Currently, I try to complete the Smallfoot case study. Once it is completed I plan to modify the abstract part, i.e. the formalisation of abstract separation logic according to what I learned:

– As motivated above, I plan to add namespaces.
– Replacing the parallel composition provided by abstract separation logic with C-like fork/join parallelism is another planned task.

In general the abstract part should become more powerful, better suited for real programming languages while still keeping as abstract as possible. More ideas for modifications may arise during the current case study, but even now, a large part of the current formalisation is due to be modified.

After finishing these modifications I plan to try some more case studies like the ones described in the introduction and iterate the process of adapting the underlying abstract semantics.

Acknowledgements

I would like to thank Matthew Parkinson, Mike Gordon, Alexey Gotsman, Magnus Myreen and Viktor Vafeiadis for a lot of discussions, comments and criticism.

References

Implementing Secure Broadcast Ambients in Isabelle using Nominal Logic

Ayesha Yasmeen and Elsa L. Gunter
Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, USA
yasmeen@uiuc.edu, egunter@cs.uiuc.edu

Abstract. In this work we present our modeling of a calculus being developed by us namely Secure Broadcast Ambients, using the nominal package for Isabelle. Our calculus is an extension of the ambient calculus where the nature of communication is broadcast within domains modeled by ambients. We allow reconfigurable configurations of communication domains, access restrictions to domains and the capability of modeling cryptographic communication protocols in broadcast scenarios.

Key words: nominal logic, process calculi, ambient calculi, broadcast, bisimulations

1 Introduction

In a world of increasing dependence on electronic communications between reconfigurable and mobile devices, there is a clear need for accurate formal systems to model these devices and their communications to facilitate guaranteeing such properties as functionality, security and privacy, and data integrity. Ambients [6], including boxed ambients [4,3], are formalisms that have been developed to model such mobile devices and their communication. Ambients have an associated topology that confine their movement and their communication options. This topology has traditionally been restricted to tree structures, and communication and movement have been restricted to adjacent ambients. The tree structure implies that an ambient can only be “in” one other ambient at a given time. This poses problems for modeling aspects of networks, such as routers. A router is most naturally modeled as being “in” multiple domains at once. Similarly, a laptop with an ethernet connection, a bluetooth connection and a dialup-modem connection, can be thought of as being “in” three different domains at once. The restriction of the topology to tree structures prevents modeling these devices that way. In this work, we loosen this constraint to allow the topology to be that of a dynamically reconfigurable directed acyclic graph, thus allowing one ambient to be in more than one other at a given time, or possibly none at all.

In theoretical models of systems, and ambients in particular, communication is often modeled using point-to-point channels. Depending on the particular calculus, many processes may have access to a given channel, but each communication will have a unique recipient. Within such frameworks, modeling broadcast and multicast communications must be done using multiple unicasts. Alternatives to this have been devised using broadcast communications. These include broadcast communication limited to a specified domain. However, in frameworks with broadcast within a domain, the domains are relatively static with only at most code moving among them. Because the code alone is mobile, it carries no identity with it, which limits the ability to concisely and accurately model the organization of the domains, and model the restriction of access to the domains by other domains. In our work, we have broadcast communication within ambients. Messages announced to the ambient are heard by the ambient and all ambients directly within it. Ambients may restrict access to themselves, and hence to the privilege of the communication within them, based on the identity of potential entrants, without requiring that their names be hidden.

Having broadcast communication, our model has the potential to naturally model communication protocols in the presence of eavesdroppers. The standard way to establish private communication using a broadcast medium is to use encryption. To facilitate reasoning about such protocols, we have enriched our language with cryptographic primitives, much in the manner of the spi-calculus [1]. To further facilitate reasoning about such protocols, which often involve information shared among certain parties, we have inductively defined
the knowledge of an ambient, in the style of [8], and give tests sufficient to prove that secrets held between interacting parties do not become a part of the knowledge of non-participating ambients. This notion of knowledge reflects the ability of an ambient to possibly synthesize new information from all information already obtained, in keeping with the Dolev-Yao model [5].

To see an example of the desirability of multiple communication domains, and the ability to span multiple domains, let us consider the scenario of a local area network comprising of a router connecting some home desktop computers and laptops to the outside world. This router is capable of directly communicating to the outside internet network and the home computers. Hence, virtually it is present in multiple communication domains simultaneously. If it is additionally a wireless router, then MAC filtering can be modeled by access restriction based on ambient identity. The ability to dynamically reconfigure the communication topology can be used to model the laptop entering the network, and later leaving it as it shuts off, or moves to another communication domain.

Continuing the example, let us consider the Ethernet local area network architecture in the home. Ethernet is built around a principle of localized broadcasting. Hence potentially, every computer in a subnet of the Ethernet can see the packets going to every other computer on that subnet. This situation has led to the advent of sniffer software, which can intercept all data on an Ethernet subnet. As a result, active attackers can use sniffing techniques to capture sensitive information and use it maliciously. As an example of how to use Secure Broadcast Ambients, we can model a scenario involving ethernet sniffer software running on a computer, Sniffer, in the Ethernet home local area network where someone wants to use another computer, Laptop, to log in to a website, giving their username and password. Unfortunately, if this information is not communicated in encrypted form, then Sniffer has every opportunity to capture it. Using the encryption primitives of Secure Broadcast Ambients, we may model protocols that allow the username and password to be communicated without being revealed to Sniffer.

We are developing a calculus Secure Broadcast Ambients, which allows broadcast communication inside regions. These regions are capable of restricting access to itself and their reconfigurable topology is allowed to form a directed acyclic graph. Mobile agents are capable of moving among the regions. The syntax of this calculus has multiple different sorts of variables, different sorts of binding constructs, many syntactic categories where several syntactic categories are mutually recursive to each other. All these features poses challenges to our goal of formalizing this calculus’s syntax and semantics in a logic like HOL. We used the nominal package of Isabelle [11] to aid us in achieving our goal. In this work we describe our experience in modeling Secure Broadcast Ambients using the nominal package of Isabelle. We intend to provide more guideline in addition to those provided in [11]. We hope that our experience will help other researchers working on modeling calculi which presents challenges similar to ours. We present the syntax of Secure Broadcast Ambients in Section 2. We then describe our implementation in Section 3. We present the formal semantics of Secure Broadcast Ambients in a condensed form while describing our implementation in that section. Finally we conclude in Section 4.

2 Syntax of Secure Broadcast Ambients

In order to define the syntax of Secure Broadcast Ambients we use the following categories of identifiers: ambient names: \( n, m \in \text{Amb} \), capability variables: \( i \in \text{CapVar} \), message variable: \( x \in \text{MessVar} \), key variables: \( k \in \text{Keys} \). The syntax of Secure Broadcast Ambients is presented in Table 1. Messages, Processes and Systems, are the main syntactic categories. Messages include message identifiers, ambient names, capabilities, key variables and data. We also allow encrypted and compound messages. Capabilities, ranged over by \( C \), can be either the capabilities for entering and exiting an ambient, capability variables or a “path”, which is a sequence of capabilities describing a mobility path. An ambient can indicate the intention to move into another ambient by in \( m \). However, this movement capability can only be successful if a corresponding co-capability is there to permit this move. The corresponding co-capability can be either in \( m \) allowing specifically \( m \) to enter, or in \( \_ \) indicating permission for any ambient. The only further restriction placed on entrance is that an ambient is not allowed to enter a descendant of itself. This interpretation of ambient movement leads to a directed acyclic graph structure for the hierarchy of ambients.
Ambient List:
\[ \mathcal{L} ::= \text{empty nil process} \mid m ; \mathcal{L} \text{ composition} \]

Capabilities:
\[ C ::= \text{in } m \text{ enter} \mid \text{out } m \text{ exit} \mid C ; C' \]

Ambient Pattern:
\[ \mu ::= \text{any ambient} \mid m \text{ ambient name} \]

Format:
\[ F ::= m : P \text{ ambient name} \mid i : P \text{ capability var} \mid 0 : P \mid \text{suc}(x) : Q \text{ natural numbers} \mid \{x\}_k : P \text{ decryption} \mid (x,y) : P \text{ pairs} \]

Actions:
\[ \pi ::= C \text{ capability} \mid \text{in } \mu \text{ allow enter} \mid (x)^m \text{ input} \mid vk.(\mathcal{S}) \text{ key restrict} \mid (M)^m \text{ output} \mid \text{create_amb}(m, P) \text{ ambient creation} \]

Processes:
\[ P, Q ::= \text{nil process} \mid P \mid Q \text{ composition} \mid \pi.P \text{ prefixing} \mid \text{cond } M \text{ is } N \text{ in } P \text{ data comparison} \mid \text{case } M \text{ of } F \text{ case analysis} \]

Systems:
\[ S ::= \text{nilsystem empty system} \mid m[P] \text{ ambient} \mid (x)^m(\mathcal{S}) \text{ broadcast receive} \mid vk.(\mathcal{S}) \text{ key restrict} \mid vm :: \mathcal{L}.(\mathcal{S}) \text{ ambient restrict} \mid S_1 \parallel S_2 \text{ parallel} \]

Table 1. Syntax of Broadcast Ambients

In Secure Broadcast Ambients, an ambient can be in multiple ambients at the same time. An ambient may even fail to be in any ambient, for example, a laptop that has been turned off. An ambient \( n \) exits from the ambient \( m \) by the \( \text{out } m \) action without requiring any permission from any other ambient, and without effecting the relationship of \( n \) to any other ambient.

Another important aspect of Secure Broadcast Ambients is that we have removed the channels used for inter-ambient communication. In our calculus the name of a parent ambient acts as the broadcast channel for both itself and its children. This way, any ambient can listen to any conversation that is going on between any of its parents and their children. Henceforth channels are synonymous with ambients.

A process can be an empty process, \( \text{nil} \). It can be a parallel composition of two processes. A process can be replicated. A process can be prefixed with some action. The actions can be to move into an ambient or to exit from an ambient, to allow entrance, to send or receive a message or to restrict keys. A process can create an ambient and at the same time define the process inside that ambient. A process can also perform matching, or case analysis on a message much in the manner of [1]. The case analysis patterns are given by the formats. The most interesting pattern is that of decryption. In this work we only consider symmetric encryption and so the decryption key is the same as the encryption key for every encrypted message. A system can be an empty system, an ambient, a system waiting to receive a message, or multiple systems in parallel. Systems can create a new key or a new ambient with a given parent list.

Ambient and key restriction, message input, ambient creation and case analysis are the binding constructs. In light of these binding constructs the free variables of messages, processes and systems are defined in the usual way.

We now show the encoding of the home router using our syntax.

\[
\text{HomeRouter}![(x)^{\text{HomeLan}}.\langle x \rangle^{\text{ISP}}.\text{nil}] \mid ![(x)^{\text{ISP}}.\langle x \rangle^{\text{HomeLan}}.\text{nil}]
\]
Its job is to capture all outgoing packets in the home network and forward them to the ISP and vice versa. Now, each computer will have to determine which of the messages (representing TCP/IP packets) arriving at HomeLan are meant for them.

3 Implementation in Isabelle

Our goal is to encode Secure Broadcast Ambients in Isabelle and to use this implementation later on to model and reason about cryptographic communication protocols in scenarios where the method of communication is broadcast. We first analyzed the calculus to determine what the most important aspects of this calculus are that should be considered before modeling in Isabelle. Examining the calculus one can observe that there are four different types of variables: message variables, capability variables, ambient names and keys. Moreover we have ambient name restrictions and key restrictions. We also have bound occurrences of all different types of variables. Also the processes are mutually recursive with some other categories. Hence we need a mechanism which will allow us to reason about $\alpha$-equivalence classes of the main syntactic categories like processes and ambient systems. We decided to use the nominal package for handling $\alpha$-equivalence classes of terms.

3.1 The first step: Atom declarations

As per the norm of using nominal package we first define the different types of data with corresponding binders that can be bound in our datatypes. They are called atoms because their internal structure is immaterial compared to their being distinguishable. From the syntax of Secure Broadcast Ambients we have observed earlier that it uses four different types of such bound entities: message variables, capability variables, ambients and keys. We use the atom var to denote a message variable, amb to denote ambients, cvar to denote capability variables and key to denote encryption key variables.

atom_decl var cvar amb key

We next show how we encoded the syntax of the calculus using nominal package.

3.2 Capabilities and Messages

We do not have any binding in the definition of messages or capabilities. Hence their declaration is quite simple. We first define the capabilities as follows:

nominal_datatype capability = IN amb
  | OUT amb
  | capas capability capability
  | CapaVar cvar

Then we define the messages.

nominal_datatype ambmsg = Var var
  | AmbM amb
  | Key key
  | Capable capability
  ... 
  | TPair ambmsg ambmsg
  | Enc ambmsg key
3.3 Processes and Systems

Processes make up the first syntactic category that has bindings. In order to associate the bindings in the actions with the corresponding bound variables in the processes we found it necessary to inline the actions in the processes. There are several noteworthy aspects of the processes. Processes have message variable bindings in their message receiving action. Processes have key restrictions and ambient restrictions. Processes have ambient binding for the new ambient creating construct. The ambient binding case is even a little bit more involved in the sense that the newly created ambient’s name has to be bound both in the process that creates it and also in the code that gets put inside it. Hence it needs to be bound in two different elements at the same time. However, as nominal datatypes do not allow a variable to be bound to a nested datatype as described in [11], we could not do that in a straight forward manner. We had to create a datatype for a pair of processes which is mutually recursive with processes. A newly created ambient’s name is now restricted to a process-pair datatype element. We made the case analyzing formats a separate datatype which is mutually recursive with the processes. The processes are defined in Isabelle as follows:

```
nominal_datatype Proc = Pnil |
| par "Proc" "Proc"
| bang "Proc"
| capa "capability" "Proc"
| entrycocapable "muamb" "Proc"
| recv "<<var>>Proc" "amb"
| send "ambmsg" "amb" "Proc"
| createamb "<<<amb>> Proc_pair"
| cond "ambmsg" "ambmsg" "Proc"
| ambcase "ambmsg" "Format"
| keypres "<<key>> Proc"
and Proc_pair = Proc_pair "Proc" "Proc"
and Format = FA "<<<amb>> Proc"
| PC "<<cvar>> Proc"
| FN "<<var>> Proc" "Proc"
| FMP "<<var>> <<<var>> Proc"
| FK "key" "<<var>> Proc"
```

Finally the ambient systems were defined as follows:

```
nominal_datatype AmbSystem =
  ...
| WholeAmb amb Proc
| Listener amb "<<var>> AmbSystem"
| ambres "<<<amb>> AmbSystem" amblist
| keyres "<<<key>> AmbSystem"
```

3.4 Substitution Functions

After defining a datatype, nominal package automatically proves lots of necessary lemmas. For a datatype of name $D$ the four most prominent ones are $D$ perm, $D$ supp, $D$ fresh and $D$ inject [11]. However, substitution of terms for atoms appearing in a new datatype is not automatically defined. Hence the step after declaring a new datatype is to define how each atom appearing in a datatype can be substituted. We defined substitution for the different types of atoms in the various datatypes that we have. Let us consider the substitution functions we defined for messages. All four types of atoms appear in messages. Hence we defined substitution for each of them. We first defined substituting a variable with a message inside a message.

```
consts substambmsgvar :: "ambmsg ⇒ var ⇒ ambmsg ⇒ ambmsg"
("._ M[:::_]" [100,100,100] 100)
We then defined substituting a capability variable with a capability in a message.

\begin{verbatim}
consts subst_ambmsgC :: "ambmsg ⇒ cvar ⇒ capability ⇒ ambmsg" ("_ MC[::= . ]") [100,100,100]
\end{verbatim}

We defined ambient and key atom substitutions similarly.

### 3.5 Proving lemmas for the user defined functions

- In the nominal logic framework, a very important concept is that of equivariance [9]. However, whereas the nominal package automatically proves equivariance lemmas for the datatypes it generates, the onus of proving the necessary lemmas for user defined functions lies on the user. After defining every new function we need to prove that the function is equivariant. As mentioned in [10] a function \( f \) is equivariant if for atom permutation \( \pi \), we have that: 
  \[ \pi \cdot (f x_1 \ldots x_n) = f(\pi \cdot x_1) \ldots (\pi \cdot x_n) \]
  where \( x_i \) are the inputs of the function \( f \).

As a result we next prove the equivariance lemmas for the variable substitution function. However one important aspect that should be remembered is that since we have four different types of atoms, we need to prove that variable substitution is equivariant for permutations of each type of atom. Hence we end up proving four equivariance theorems for each variable substitution function.

As an example, for the function \( M[::=.] \), which substitutes a variable with a message in a message, we have four equivariance lemmas looking like:

\begin{verbatim}
lemma subsvar_var_eqvt[eqvt, simp]:
\end{verbatim}

\begin{verbatim}
shows \( \pi \cdot (m \ M[ x ::= t]) = ((\pi \cdot m) \ M[ (\pi \cdot x) ::= (\pi \cdot t)]) \)
\end{verbatim}

where \( \pi \) is message variable, capability variable, ambient or key permutations respectively. Every such equivariance theorem should be given the equivariant attribute \texttt{eqvt}.

- We also defined support for the variable substitution function. Basically it says that the free message variables of a message \( m \) after substituting a variable \( x \) with a message \( t \) is a subset of the free variables of \( m \) and \( t \) sans \( x \).

\begin{verbatim}
supp(m \ M[x:=t]) \subseteq (supp(m)-\{x\}) \cup supp(t):: (var set)
\end{verbatim}

Here we need to explicitly mention that the support is being defined for \texttt{var} atoms. Here support lemma is only needed for the type of atom that is being replaced in a datatype.

- We also proved freshness lemmas for the substitution functions.

In a nutshell the pattern that we followed were:

- for each new datatype do the following:
  - define substitution for every type of atom appearing in the datatype
  - define equivariance for every type of atom for each substitution function.
  - define support for the substitution function for the type of atom that is being replaced
  - define freshness for the substitution function

This procedure has to be followed for every datatype and for every atom if the datatype is to be used later in some other function. For example while defining substitution in processes we needed to reason about substitution in capabilities, messages and other nominal datatypes appearing in the processes. Hence substitution for these datatypes have to have been properly defined and their equivariance should already have been proved.

### 3.6 Substitution in Processes

Substitution of terms for atoms in processes were challenging for many reasons. A process has all different types of atoms appearing in it and also it is mutually recursive with two other nominal datatypes. It also contains various bindings. Hence defining substitution and proving the required lemmas about them have a slightly different flavor.
Message variable substitution in processes is defined as follows:

```plaintext
consts subst_proc :: "Proc ⇒ var ⇒ ambmsg ⇒ Proc" ("_ P[::= _]" [100,100,100] 100)
consts subst_form :: "Format ⇒ var ⇒ ambmsg ⇒ Format" ("_ F[::= _]" [100,100,100] 100)
consts subst_Proc_pair :: "Proc_pair ⇒ var ⇒ ambmsg ⇒ Proc_pair" ("_ PP[::= _]" [100,100,100] 100)

nominal_primrec
(* "_ P[::= _]" is defined here *)
"Pnil P[ x ::= t ] = Pnil"
...
" x1 # (m, x, t)⇒ (recv x1 P m) P[ x ::= t ] = (recv x1 (P P[ x ::= t ] m))"
" (send m1 m2 P[ x ::= t ] = (send (m1 M[ x ::= t ]) m2 (P P[ x ::= t ])))"
"m1 # (x, t)⇒ (createamb m1 PP ) P[ x ::= t ] = (createamb m1 (PP PP[ x ::= t ]))"
...
" (ambcase m f1) P[ x ::= t ] = (ambcase (m M[ x ::= t ]) (f1 F[ x ::= t ]))"
"[| k # (x,t)|⇒ ( keypres k P) P[x::=t] = (keypres k (P P[x::=t]))"

and (* "_ PP[::= _]" is defined here *)
"(Proc_pair P1 P2) PP[x ::=t] = (Proc_pair (P1 P[x::= t] ) (P2 P[x::= t ]))"

and (* "_ F[::= _]" is defined here *)
"m # (x, t)⇒ subst_form (FA m P) x t = (FA m (P P[ x ::= t ])))"
"c # (x,t)⇒ (FC c P1) F[x ::= t] = (FC c (P1 P[ x ::= t ])))"
"x1 # (P1, x, t)⇒ (FN x1 P2 P1) F[x ::= t] = (FN x1 (P2 P[ x ::= t ]) (P1 P[ x ::= t ])))"
"[|x1 # (x2, x, t); (x2 # (x1,x,t)|⇒
(FMP x1 x2 P) F[x::=t]= (FMP x1 x2 (P P[x::=t])))"
" x1 # (k, x, t)⇒(FK k x1 P) F[x::= t] = (FK k x1 (P P[x::=t]))"

The reason for displaying such a big chunk of code was to illustrate various points.

- Defining substitution for a mutually recursive nominal datatype requires substitution functions for all the mutually recursive elements to be defined together. The reason being that the recursion combinator expects an equation for every term constructor.
- Special care needs to be taken for the constructs with bound atoms. The binders must be fresh in the variable that is being substituted, the term with which it is being substituted and any other bound atoms in that construct. If the freshness constraints are not provided properly then definition of the substitution function will result in subgoals which are either false or vacuous or hard to prove. For example, the rule for substitution in the new ambient creation construct is: "m1 # (x, t)⇒ (createamb m1 PP ) P[ x ::= t ] = ...". Here the bound ambient atom m1 has to be fresh for both x and t. Also if there are multiple bound atoms in a term they have to be distinct from each other. For example in the rule "[|x1 # (x2, x, t); (x2 # (x1,x,t)|⇒
(FMP x1 x2 P) F[x::=t]= (FMP x1 x2 (P P[x::=t])))" where we match up a pair of messages, both the atoms x1 and x2 are bound in P. Hence the condition for x1 is x1 # (x2, x, t) and similarly for x2.
- Nominal package needs a lot of properties of each recursive function operating on nominal datatypes to be proved before it can be used. The properties mainly deal with the finiteness of the support of
the function and its parameters and freshness constraints. However, the list of subgoals thrown at the
user can be non-trivial. For functions like variable substitution in messages we find that the subgoals
are trivial. They merely ask us to prove $\text{True}$. Repeated application of the rule $\text{TrueI}$ takes care of
all of them. It would have been simpler if this was done automatically by the nominal package. The
substitution for messages is easier as it does not have binders or mutual recursion. But the substitution
for processes is not easy and the nominal package gives us a total of 255 subgoals. We describe the tactics
used by us to dispose of them next.

- A lot of the subgoals returned by the substitution in processes can be gotten rid of by simplifying with
equivariance lemmas and other lemmas like one type of atom being fresh for other types of atoms and
so on. Then the the subgoals which needed to prove some finiteness of supports were taken care of by
the tactic $\text{finite guess}$. Then the rest of the goals were about proving freshness conditions which were
easily taken care of by the tactic $\text{fresh guess}$. However we must assert that for these tactics to succeed
properly, you have to have proven equivariance, support lemmas and freshness lemmas for each and every
function used in the definition of the substitution function. For example, consider substituting in the
case of sending a message over an ambient given by

$$(\text{send } m1 \text{ m2 P} \ [ \ x ::= t \ ] = (\text{send } (m1 \ M[ \ x ::= t \ ]) \ m2 \ (P \ P[ \ x ::= t \ ])))$$

Here we substitute the variable $x$ with the message $t$ inside the message $m1$ that is being sent over
the ambient $m2$. Hence we use the appropriate substitution function $\_ M[\_ ::=\_ ]$. If all the necessary
lemmas for that function are not proved earlier then the substitution for processes gives us subgoals that
can be harder to prove.

- After defining a function we need to prove equivariance, support and freshness results for that function.
For the functions dealing with simpler nominal datatypes like messages which do not have any bound
variables, defining these lemmas are very easy. However, for processes it was a little bit more complex. For
example, for equivariance of processes we need to define what it means for all of the mutually recursive
elements together in the following format:

$$(\pi \cdot (P \ P[ \ x ::= t \ ]) = (\pi \cdot P) \ P[ \ (\pi \cdot x) ::= (\pi \cdot t) ])$$

and

$$(\pi \cdot (pp \ PP[ \ x ::= t \ ]) = (\pi \cdot pp) \ PP[ \ (\pi \cdot x) ::= (\pi \cdot t) ])$$

and

$$(\pi \cdot (f \ F[ \ x ::= t \ ]) = (\pi \cdot f) \ F[ \ (\pi \cdot x) ::= (\pi \cdot t) ])$$

We have four such lemmas where $\pi$ is $\text{var prm}$, $\text{cvar prm}$, $\text{amb prm}$ and $\text{key prm}$ respectively.
An example freshness lemma is given here:

$$(\lfloor x \not\in P \rfloor \Rightarrow (P \ P[ \ x ::= t \ ] = P))$$

and

$$(\lfloor x \not\in PP \rfloor \Rightarrow (PP \ PP[ \ x ::= t \ ] = PP))$$

and

$$(\lfloor x \not\in F \rfloor \Rightarrow (F \ F[ \ x ::= t \ ] = F))$$

where $x$ is of type $\text{var}$.

- While proving the equivariance, support and freshness results for functions substituting atoms in pro-
cesses, we needed to induct on the processes. We needed to use an inductive hypothesis that will induct
on all the related mutually recursive datatypes together. In our case, we have $\text{Proc}$, $\text{Proc pair}$ and
$\text{Format}$ which are mutually recursive. The inductive hypothesis was named:

$\text{Proc Proc pair Format inducts}$

Notice that you have to use $\text{inducts}$ not $\text{induct}$. Hence the rule is that for a mutually recursive datatype $P$
where $P_1, P_2, \ldots, P_n$ are mutually recursive, the inductive hypothesis looks like: $P_1, P_2, \ldots, P_n, \text{inducts}$

We had to follow the above mentioned tasks for all different types of atoms appearing in a datatype. As a
result, we ended up defining four different substitution functions for the processes. They were very similar
in appearance. However, we always had to take care so that required freshness constraints are properly
provided, so that there is no inadvertent free variable capture and that we do not attempt to substitute a
bound variable.

3.7 Formal Semantics

After we have defined our datatypes with binders where necessary, defined substitution functions for all of
them for all atoms as necessary and proven necessary lemmas for these functions, we find ourselves with the
appropriate background for using the machinery to encode the formal semantics of the calculus.
(StrRepPar) !P ≡ P | !P
(StrNilProc) m[|nil|] ≡ nilsystem
(StrPathPref) (|C1|; C2).P ≡ C1.C2.P
(StrSysPar) m[P1 | P2] ≡ m[P1].P2
(StrCond) cond M is M in P ≡ P
(StrAmbMsg) case m of n : P ≡ P[m/n]

... (StrProcSys) P ≡ Q ⇒ m[P] ≡ m[Q]
(StrNilProc) n[create_amb(m, P), Q] ≡ vm :: empty.
(StrSysPar) m[P1 | P2] ≡ m[P1].P2
(StrNilProc) m[|nil|] ≡ nilsystem
(StrBrdcstListen) νm.νnilsystem ≡ nilsystem

Table 2. Structural Equivalence

Structural Equivalence: Structural equivalence is defined for each of processes and systems. It is the smallest congruence containing the rules in Table 2, closed under alpha equivalence (where ν, message receipt, ambient creation, and case analysis are the binding constructs), and the associativity and commutativity of parallel composition of each of processes and systems, with nil and nilsystem as the respective identities. In the table we use u to denote m :: L or k.

The last three rules in Table 2 are the rules that enable broadcast communication. The rule (StrBrdcstListen) allows an ambient to lift the receive action on a particular broadcast channel of a process within it up to the level of the ambient system. The rule (StrComblListen) can then be used to combine the listening systems. It is the principal rule that is used to model broadcast systems. This rule combines multiple ambients listening on the same channel so that later on only one reaction is needed to send a message simultaneously to all the ambients listening on this ambient. In addition to being able to send a message to multiple parties simultaneously, in a broadcast scenario, the broadcaster can send out a message even if no one is listening on the broadcast channel being used. The rule (StrNoListen) enables us to model this scenario. Structural equivalence of two ambient systems are provided at the system level and the process level. First we encode the broadcast channel being used. The rule (StrNeutral)

inductive StructEquivProc :: "Proc * Proc ⇒ bool" where
nilproc[intro!]:"StructEquivProc((par Phil P), P)"|

... ambcompcase[intro!]: "m2 # m1 ⇒ StructEquivProc ( ambcase (AmbM m1) (FA m2 P)), (P PA[ m2 ::= m1 ]))" |

... The inductive definition also has some rules similar to substitution function definition. For the cases where binders are handled, appropriate freshness constraints have to be inserted [11]. Inductively defined relations also need to be equivariant. Nominal package provides an easier way of determining whether an inductively defined relation is equivariant or not. They provide a simple tactic for performing this task. The name of the tactic is "equivariance". In order to prove that a relation R is equivariant, after inductively defining a relation we simply need to type equivariance R. In our case, the command we used was equivariance StructEquivProc.

Similar to proving equivariance of functions, we need to have that all other user defined function used in the inductive definition be equivariant themselves. For example in the ambcompcase case above, if the equivariance proving lemma for the ambient substitution function _PA[:::_] is removed, then the nominal package will give an error message saying that it failed to prove that
Exit:

\[
\begin{align*}
\Delta \not\# n \in \mathcal{L}_n & \quad \mathcal{L}_n \not\# \Pi \triangleright n[\text{out } m . P] \\
\Delta \not\# n \in \mathcal{L}_n & \quad \mathcal{L}_n \not\# \Pi \triangleright P
\end{align*}
\]

\[
\begin{align*}
\Delta \triangleright S_1 \equiv \Delta_1' \triangleright S_1' & \quad \Delta_1' \triangleright S_1' \to \Delta_2' \triangleright S_2' \\
\Delta_2' \triangleright S_2' \equiv \Delta_2 \triangleright S_2 & \quad \Delta_1 \triangleright S_1 \to \Delta_2 \triangleright S_2
\end{align*}
\]

FormationEquiv:

<table>
<thead>
<tr>
<th>KeyRestrict:</th>
<th>AmbRestrict:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Delta \triangleright S_1 \to \Delta_1' \triangleright S_2), (k \notin \text{fv}(S_1)), (k \notin \text{fv}(S_2))</td>
<td>(\Delta \not# n \in \mathcal{L} \triangleright S_1 \to \Delta_1' \not# n \in \mathcal{L}' \triangleright S_2)</td>
</tr>
<tr>
<td>(\Delta \triangleright \nu k.S_1 \to \Delta_1' \triangleright \nu k.S_2)</td>
<td>(\Delta \triangleright \nu n :: \mathcal{L}.S_1 \to \Delta_1' \triangleright \nu n :: \mathcal{L}'.S_2)</td>
</tr>
</tbody>
</table>

Table 3. Unlabeled Transition System

\[\text{StructEquivProc}\] is equivariant. We encode structural equivalence of systems using the relation \(\text{StructEquivSys}\) (omitted here).

**Unlabeled Transition System**: We impose topological structure on the ambients using “configurations”. The configurations keep track of the topological layout of the ambients in a system. Roughly, for every ambient in a system, it lists the ambients it is in. A configuration is a list of pairs where each pair has the name of an ambient and the list of ambients it is in. The pair of a system, \(S\) and its topological structure that is its configuration, \(\Delta\), is called a formation denoted by \(\Delta \triangleright S\) and implemented as “ConfigSys” in Isabelle.

\[
\text{datatype Config} = \text{EmptyConfig} \\
| \text{Conf \ amb amblist Config} \\
\text{datatype ConfigSys} = \text{CS Config AmbSystem}
\]

We provide part of the unlabeled transition system in Table 3. We encoded the unlabeled transition system as follows:

\[
\text{inductive UTS :: } \text{"ConfigSys * ConfigSys \Rightarrow bool" where} \]

\[
\text{xsn.exit[intro]:} \quad "(\text{InL (findlist C n) m}) \Rightarrow \text{UTS}((\text{CS C (WholeAmb n (capa (OUT m) P))),(CS (delfromconf C n m ) (WholeAmb n P)))" |} \\
\text{xsn.formequiv[intro]:} \quad "[| UTS((\text{CS C1 S1}),(\text{CS C1a S1a})); UTS ((\text{CS C1a S1a}),(\text{CS C2a S2a})); \text{StructEquivCS((CS C2a S2a),(CS C2 S2))}] \Rightarrow \text{UTS}((\text{CS C1 S1}),(\text{CS C2 S2}))" |
\]

In this definition the relation \(\text{StructEquivCS}\) stands for the relation defining structural equivalence for formations. We omit the definition of structural equivalence for formations in this work. The rule \(\text{xsn.exit}\) stands for the Exit transition rule in Table 3. If an ambient \(n\) wants to exit another ambient \(m\), then we first check that in the configuration \(C\) (or \(\Delta\) in the Exit rule) asserts that the ambient \(n\) is actually inside ambient \(m\). After ambient \(n\) exits ambient \(m\), the configuration \(C\) is updated by deleting ambient \(m\) from ambient \(n\)’s parent list.

**Observational Equivalences**: Observational (barbed) equivalences focus only on the observable actions and do not consider the messages being exchanged in the transitions. We denote a formation \(\Delta \triangleright S\) exhibiting a barb \(\xi\) by \(\Delta \triangleright S \downarrow \xi\). We present some of the barbs in Table 4. The barbs exhibited by a formation is defined inductively in Isabelle as follows:

\[
\text{inductive showsbarb :: } \text{"ConfigSys * \text{barb } \Rightarrow bool" where} \]

\[
\text{barbsend[intro]:} \quad "(\text{InL (ALC n (findlist c n) m}) \Rightarrow \text{showsbarb} ((\text{CS c (WholeAmb n (send msg1}}
\]

132
Barb Send:
\[ m \in (\{n\} \cup \Delta(n)) \]
\[ \Delta \vdash n[(M)^\mu]P \downarrow \text{send } m \]

Barb In:
\[ m \in \text{dom}(\Delta) \land m \notin L_n \]
\[ \Delta \vdash \Pi[n]m.P \downarrow \text{in}(n,m) \]

### Table 4. Barbs

<table>
<thead>
<tr>
<th>Barb Send:</th>
<th>Barb In:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ m \in ({n} \cup \Delta(n)) ]</td>
<td>[ m \in \text{dom}(\Delta) \land m \notin L_n ]</td>
</tr>
<tr>
<td>[ \Delta \vdash n[(M)^\mu]P \downarrow \text{send } m ]</td>
<td>[ \Delta \vdash \Pi[n]m.P \downarrow \text{in}(n,m) ]</td>
</tr>
</tbody>
</table>

\[ \Delta \vdash \nu k.S \downarrow \xi \]
\[ \Delta_1 \vdash S_1 \downarrow \xi \quad \Delta_2 \vdash S_2 \downarrow \xi \]
\[ \Delta \vdash (S_1 \parallel S_2) \downarrow \xi \]

In order to determine what secrets a spy has gleaned in a system, we want to reason about the knowledge of
an ambient in an ambient system. The knowledge of an ambient in a system is a set of messages. It is calculated
by syntactically traversing a system and determining what information is available to a particular
ambient in that system. Knowledge may be finite or it may be infinite. While laying out the foundation
for providing the security lemma, we realized that there are some lemmas about freshness in sets that are
missing in the nominal package. Generally speaking, if we have multiple atoms say \( a, b, c, \ldots \), then it should
be the case that \( a \# (X:: b \text{ set}) \), \( a \# (X:: c \text{ set}) \) and so on. However, there is no such generic lemma in the
nominal package which we felt were due to the reason that this requires reasoning about a set which
may well be infinite, and in general providing the machinery automatically for functions and infinite sets are
not easy to perform automatically. Still we feel that lemmas like these should be included in the nominal
package. Knowledge of an ambient \( m \) in a given ambient system is defined (in part) as follows:

```
| function knowledge :: "AmbSystem \Rightarrow amb \Rightarrow (ambmsg set) \Rightarrow (ambmsg set)" where
| "knowledge NilSystem m I = {} "
| | "knowledge (WholeAmb n P) m I = (if (m=n) then ((procinfo P I) \cup (AmbM m-I)) else "
```

Here \( \text{procinfo} \) is a function which calculates the information contained in a process. We are now working on
proving a secrecy theorem which tries to figure out under which conditions elements from a set of secrets will not end up in the knowledge of some spy ambients. We have defined the function that calculates knowledge of ambients in a system.

4 Conclusion and Related Work

We modeled a calculus namely Secure Broadcast Ambients which are capable of modeling cryptographic communication protocols of mobile agents where the nature of communication is broadcast within a domain. Our calculus allows a directed acyclic graph topology of the location of our agents, hence our topology is more flexible than the usual tree structured one. Our domains or systems are also capable of restricting access to themselves. Our focus was on delineating the steps we had to take and hurdles that we had to overcome to implement the calculus using nominal package. We encoded structural equivalence, unlabeled transition system, behavioral equivalence and congruence. We introduce the idea of knowledge of an ambient so that we can reason about how much information untrusted ambients have acquired in a given ambient system. Our future goal is to prove a secrecy theorem which will roughly give the conditions under which processes that do not reveal secrets to untrusted agents can be indistinguishable. Many work has been done on implementing various calculi in Isabelle using the nominal package. Bengtson et al. has formalized the π-calculus using the nominal package [2]. Kahsai et al. has formalized the spi-calculus using the nominal datatype [7]. The examples directory of the nominal package presents various theories. Theories like Class.thy which implements term calculus was very beneficial to us as it deals with multiple atoms. We were also helped a lot by the nominal mailing list regarding issues of mutually recursive nominal datatypes.

References

Author Index

Chalin, Patrice .................. 1
Conchon, Sylvain ................. 64
Gawanmeh, Amjad ................ 9
Gunter, Elsa ..................... 123
Hasan, Osman ................... 21
Hiss, Harald ..................... 33
Huffman, Brian .................. 45
James, Perry R. .................. 1
Karabotsos, George ............. 1
Krauss, Alexander ............... 57
Lescuyer, Stéphane ............. 64
Mokhov, Serguei ................. 76
Myreen, Magnus O. .............. 99
Paquet, Joey ..................... 76
Spichkova, Maria ............... 104
Tuerk, Thomas .................. 116
Yasmeen, Ayesha ................. 123