Serge Autexier,
Heiko Mantel (Eds.)

Second Verification Workshop
VERIFY'02

affiliated with the 18th Conference
on Automated Deduction (CADE)
at FLoC'02

July 25–26, 2002
Copenhagen, Denmark

Preface

Traditionally, the verification of system properties has been one of the main areas of application for automated theorem proving. On the one hand side, the formal development of safety and security critical systems creates numerous deduction problems which are not only interesting and challenging but also of practical relevance. On the other hand, automated theorem proving offers the means to reduce the development burden in formal developments, thus making them feasible.

The aim of the verification workshop is to bring together people who are interested in the development of safety and security critical systems, in formal methods in general, in automated theorem proving, and in tool support for formal developments. The emphasis of the workshop is on the identification of open problems and the discussion of possible solutions under the theme

What are the verification problems? What are the deduction techniques?

VERIFY'02 was held in connection with the *Conference on Automated Deduction (CADE)* at *FLoC* in Copenhagen, Denmark. This year, 16 regular papers were submitted from which 11 were selected for presentation at the workshop. These 11 accepted papers and an abstract of the invited talk by Fabio Massacci about *"Formal Verification of SET by Visa and Mastercard: Lessons for Formal Methods in Security"* are contained in this report.

The special focus of this years workshop was the application of formal methods to issues in computer security. Submissions in this area were especially encouraged and joint submissions to VERIFY and to the workshop on Foundations of Computer Security (FCS) were possible. Almost two thirds of the accepted papers were security-related while the remaining papers covered other topics. Hence, the final program of VERIFY nicely reflected the special emphasis on security without neglecting other important topics.

We would like to thank several people who helped us in the organization of this workshop. First of all, many thanks to all program committee members for their support and productive collaboration. Many thanks to the organizers of FLoC'02 and CADE'02 for their support in the organization and realization of this workshop, especially, to the FLoC Workshop Chair Sebastian Skalberg, the CADE Workshop Chair Mateja Jamnik, and the CADE Conference Chair Reiner Hähnle, for answering various curious questions that occurred to us during the preparation of this workshop. Last but not least, many thanks to all authors who submitted papers to this workshop.

Serge Autexier
Heiko Mantel
Workshop Organizers

Workshop Committee

Program Co-Chairs & Organizers

Serge Autexier, University of Saarbrücken, Germany
Heiko Mantel, German Research Center for Artificial Intelligence, Germany

Program Committee

David Basin, University of Freiburg, Germany
Iliano Cervesato, ITT Industries, USA
Riccardo Focardi, University of Venezia, Italy
Rainer Hähnle, Chalmers University, Sweden
Nevin Heintze, Agere Systems, USA
Andrew Ireland, Heriot-Watt University, UK
Deepak Kapur, University of New Mexico, USA
Christoph Kreitz, Cornell University, USA
Fabio Martinelli, CNR Pisa, Italy
Fabio Massacci, University of Trento, Italy
Catherine Meadows, Naval Research Lab, USA
Steve Schneider, Royal Holloway, University of London, UK

Table of Contents

Security Protocols

Specification and Verification

Panel

Session:
Invited Talk

Formal Verification of SET by Visa and Mastercard: lessons for formal methods in security

Fabio Massacci

Dipartimento di Informatica e Telecomunicazioni
Universitá di Trento - Italy
http://www.ing.unitn.it/~massacci - massacci@ing.unitn.it

Abstract. The Secure Electronic Transaction (SET) protocol has been proposed by a consortium of credit card companies and software corporations to secure e-commerce transactions. When the customer makes a purchase, the SET dual signature guarantees authenticity while keeping the customer's account details secret from the merchant and his choice of goods secret from the bank.
SET verification has alwasy been a holy grail for security verification and many papers do conclude with "and this technique can be applied to SET" and yet the forthcoming application is not so forthcoming...
In this talk, I report the results of the verification efforts on the SET protocol, a joint work with G. Bella and L. Paulson from the University of Cambridge. In a nutshell, we proved that the protocol is reasonably secure. by using Isabelle and the inductive method we showed that the credit card details do remain confidential and customer, merchant and bank can confirm most details of a transaction even when some of those details are kept from them.
And now, the question come: you verified SET, so what?
What can we learn for this verification effort? Are there lessons for security design? Which security designs are easier to verify? What kind of techniques and tricks are necessary? What do we need to scale so that security verification can become an easier task? I will give a personal perspective on the problem.

1 Why SET (and real industrial protocol) are hard for security verification

The last years have seen a substantial progress in the formal verification of security protocols. Detailed analysis of cryptographic primitives, verification of Internet standards, and substantial progress in the automation of model-checking and theorem-proving procedures for security verification have boosted a field which outsiders believe populated by "Yet-Another-Weakness-of-Needham-Schroeder" papers.

Though protocols like Kerberos IV [3], the Internet Key Exchange protocol [10], the Cybercash protocol [5], the TLS/SSL protocol [12], the Cardholder

Registration Phase of SET[2] all yielded to automatic or semi-automatic tools, full verification of SET (the Secure Electronic Transaction protocol by Visa and Mastercard) has remained out of reach.

Lots of researchers have worked on the problem: for instance Meadows and Syverson [11] have proposed a language for describing SET specifications but have not actually verified the protocol. Kessler and Neumann [4] have extended an existing belief logic with predicates and rules to reason about *accountability*. Although accountability is not a stated goal of SET, it is clearly desirable. They concentrate upon the Merchant's ability to prove to a third party that the Order Information originated with the Cardholder. Using the calculus of the logic, they conclude by pen and paper that the goal is met, so the Cardholder cannot repudiate the transaction. Equivalently, we have proved that the dual signature being in the traffic implies that the Cardholder sent it. Stoller [14] has proposed a theoretical framework for the bounded analysis of e-commerce protocols but has only considered an overly simplified description of the payment phase of SET. Hui and Lowe [5] have proposed a general theory to transform a complex protocol into a simpler protocol while preserving any faults. However, they limited their actual analysis to the Cybercash protocol.

Why is SET such a challenge for formal verification? The first obstacle is its documentation [6–9] which takes over 1000 pages. The second obstacle is its protocol. Academic protocols are typically short, straight-line programs; they seldom go beyond two levels of encryption and generate few secrets. Even more sophisticated protocols with more complex goals like Optimistic Fair Exchange [1] or Group Protocols can be described into few pages. Internet protocols such as IKE and TLS use cryptography rather sparingly compared to SET. SET has many features that make its verification hard:

- multiple nested encryptions and several message fields which require abbreviations, make the manual unwinding of the specifications impossible and restrict analysis to tools supporting equational reasoning;
- ubiquitous generation of random numbers and keys hampers the usual model-checking technique to limit the state space (limiting different keys and nonces to an handful) as it would not even allow a single execution to complete, let alone two or more parallel ones;
- many alternative protocol paths make it impossible to single out the few key roles used either by manual analysis (as in the strand space model) or by model-checkers to restrict the search space;
- many different cryptographic algorithms (xor, symmetric keys, hash functions, public key encryption and digital signatures, random padding) that makes at list suspicious the assumption of perfect cryptography

Are they caused by bad design? Though some security expert may claim that SET is badly designed because it was designed by a committee, other will rightly claim that many of these features are actually needed in any practical protocol. For sake of example take the presence of optional protocol paths: they are necessary in any practical scenarion in which we remember that the task of security

protocol is first doing business, second doing it securely. Security-aware customers may have pre-registered with a financial institution and thus secured their credit cards against the merchant's eyes. Other customers may decide to trust the merchant and thus be content with a transaction secured against the outside world. From a merchant's perspective, all customers should able to conclude a purchase, whether they bothered to pre-register or not.

The complex structure of SET makes it a benchmark for security protocol design and verification, whether or not it will be a commercial success. It is a baseline test to check whether one can scale up to the point when direct manual analysis no longer work, and human intuition can no longer guide fine-state methods in getting the right "configuration" for finding bugs.

We succeeded in analyzing an abstract, but still highly complex, version of the SET purchase protocols. The difficulty consisted in digesting the specification and scaling up. This is a major result: our methods scale to a level of complexity where intuition falters but... we found out that our method, based on human interaction with a semi-automatic but powerful prover, has reached a point where the sheer complexity of the proofs, the size of what must be read (possible protocol configurations that must be ruled out) will require further advances to scale further.

In the talk I will give a personal perspective on the problem: what can we learn for this verification effort? Are there lessons for security design? Which security designs are easier to verify? What kind of techniques and tricks are necessary? What do we need to scale so that security verification can become an easier task?

References

1. N. Asokan, M. Schunter, and W. M. Optimistic protocols for fair exchange. In *Proc. of the 4th ACM Conf. on Comm. and Comp. Sec. (CCS-97)*, pages 7–17. ACM Press and Addison Wesley, 1997.
2. G. Bella, F. Massacci, L. C. Paulson, and P. Tramontano. Formal verification of cardholder registration in SET. In F. Cuppens, Y. Deswarte, D. Gollman, and M. Waidner, editors, *Computer Security — ESORICS 2000*, LNCS 1895, pages 159–174. Springer, 2000.
3. G. Bella and L. C. Paulson. Kerberos version IV: Inductive analysis of the secrecy goals. In Quisquater et al. [13], pages 361–375.
4. V. Kessler and H. Neumann. A sound logic for analysing electronic commerce protocols. In Quisquater et al. [13].
5. G. Lowe and M. L. Hui. Fault-preserving simplifying transformations for security protocols. *J. of Comp. Sec.*, 9(3-46), 2001.
6. Mastercard & VISA. *SET Secure Electronic Transaction: External Interface Guide*, May 1997. Available electronically at `http://www.setco.org/set_specifications.html`.
7. Mastercard & VISA. *SET Secure Electronic Transaction Specification: Business Description*, May 1997. Available electronically at `http://www.setco.org/set_specifications.html`.

8. Mastercard & VISA. *SET Secure Electronic Transaction Specification: Formal Protocol Definition*, May 1997. Available electronically at `http://www.setco.org/set_specifications.html`.
9. Mastercard & VISA. *SET Secure Electronic Transaction Specification: Programmer's Guide*, May 1997. Available electronically at `http://www.setco.org/set_specifications.html`.
10. C. Meadows. Analysis of the Internet Key Exchange protocol using the NRL Protocol Analyzer. In *SSP-99*, pages 216–231. IEEE Comp. Society Press, 1999.
11. C. Meadows and P. Syverson. A formal specification of requirements for payment transactions in the SET protocol. In R. Hirschfeld, editor, *Proceedings of Financial Cryptography 98*, volume 1465 of *Lecture Notes in Comp. Sci.* Springer-Verlag, 1998.
12. L. C. Paulson. Inductive analysis of the internet protocol TLS. *ACM Trans. on Inform. and Sys. Sec.*, 2(3):332–351, 1999.
13. J.-J. Quisquater, Y. Deswarte, C. Meadows, and D. Gollmann, editors. *Computer Security — ESORICS 98*, LNCS 1485. Springer, 1998.
14. S. D. Stoller. A bound on attacks on payment protocols. In *Proc. 16th Annual IEEE Symposium on Logic in Computer Science (LICS)*, June 2001.

Session:
Applications of ATP in Verification

Application Specific Higher Order Logic Theorem Proving*

Daniel Kroening
Computer Science Department
Carnegie Mellon University
kroening@cs.cmu.edu

Abstract

Theorem proving allows the formal verification of the correctness of very large systems. In order to increase the acceptance of theorem proving systems during the design process, we implemented higher order logic proof systems for ANSI-C and Verilog within a framework for application specific proof systems. Furthermore, we implement the language of the PVS theorem prover as well-established higher order specification language. The tool allows the verification of the design languages using a PVS specification and the verification of hardware designs using a C program as specification. We implement powerful decision procedures using Model Checkers and satisfiability checkers. We provide experimental results that compare the performance of our tool with PVS on large industrial scale hardware examples.

1 Introduction

1.1 Challenge

Formal verification of complex systems such as hardware and software designs is the major thrust within the theorem proving community in the past years. Formal verification is applied in a wide range of domains. First of all, in the domain of safety critical systems a full proof of correctness is most desirable. Examples of live-critical embedded systems include medical devices and controllers in the avionics or automotive industry. Because of the high cost of design faults, theorem proving is already common in this area. Examples include the formal verification of a fault tolerant communication bus protocol [PSvH99, Pfe00] using PVS or the verification of tools for train borne control software systems [BT00] using ACL2 [KM96].

In case of the chip industry, design faults are expensive due to shortening time-to-market. A well known example is the bug in Intel's Pentium floating point unit [V. 95]. Despite of the progress of symbolic Model Checking, state of the art microprocessors are still too complex for completely automated formal verification methods. Theorem proving is currently the only technique known that is able to handle designs of this complexity. Examples of theorem proving within the microprocessor industry include the work of David Russinoff [Rus00b] on the correctness of the floating point units of the AMD Athlon processor series using ACL2 and of John Harrison on Intel's Merced [Har99] using HOL [CGM86].

However, theorem proving has not yet become an integral part of the design process. Main obstacles are the high amount of manual work required by theorem proving systems. We discuss two issues that are part of this problem. The first issue is the language that is used for design, specification and verification, and the second is automation of the proof itself.

Design Language The formal verification of hardware and software designs usually includes that the original design, given in a programming or hardware description language, first has to be translated to the native language of the theorem prover. If done manually, this translation process is error-prone, since while formalizing the design, the person doing the translation is likely to translate the "desired behavior" instead of a bug.

*This research was sponsored by the Semiconductor Research Corporation (SRC) under contract no. 99-TJ-684, the National Science Foundation (NSF) under grant no. CCR-9803774, the Office of Naval Research (ONR), the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796, and by the Defense Advanced Research Projects Agency and the Army Research Office (ARO) under contract no. DAAD19-01-1-0485. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of SRC, NSF, ONR, NRL, DOD, ARO, the U.S. government or any other entity.

The design is often harder to understand after translation into the language of the theorem proving system. This holds particularly for sequential programming languages, such as ANSI-C or Java. In case bugs are found while trying to prove the correctness of the design, one is required to understand the nature of the bug given an output in the native language of the theorem prover. The bug has to be fixed in the design language and then the translation has to be repeated.

An alternative approach is to write the design in the high level, native language of the theorem proving system. The design is later on obtained by translating or compiling the high level languages into the design language. However, this approach is not accepted in many areas because of existing code, that is to be re-used, and because of low efficiency of the translated code. This is a particular problem for embedded systems designers, where code size and speed are crucial for the total cost of the product.

We therefore think that the language a theorem prover uses is of high importance for the acceptance of theorem provers as a design tool. The theorem prover should accept both hardware and software designs in the original, low level design language. This allows efficient designs and formal verification without translation into a specific theorem proving language. Feedback of any kind, e.g., subgoals in case of an interactive theorem prover, or about bugs in the design, should be given in the original language as well.

Specification Language While the design language is intended for efficient compilation of the hardware or software product, the specification, in contrast, should be given in a concise high level language. In [Gor85], Gordon motivates the use of higher-order logic for the specification of hardware. We believe that the specification language should not be application specific. Higher-order logic ensures that the specification language applies to the widest range of applications. In [Rus97], Rushby describes the advantages of sub-typing, as supported by PVS, for writing easily understandable and concise specifications.

Proof Automation Most large practical verification problems in the hardware and software domain are solved compositional, i.e., the design is manually divided into smaller sub-problems that are then verified independently. This allows reducing the large problem into a number of problems that are small enough to be solved by efficient automated decision procedures. These decision problems are often very application specific. However, general purpose theorem proving systems often lack efficient decision procedures for these problems. They then have to be solved manually, which is an unnecessary burden.

Both for low level programming languages, such as ANSI-C and Java, and, obviously, hardware design languages, decision support for bit vector arithmetic is highly desirable. This includes operations such as addition, multiplication, shifting and bit-wise operations on fixed-length bit vectors. However, in most existing theorem proving systems automated decision support for bit vector arithmetic is neglected.

The correctness proof of control logic of hardware designs is often very tedious using theorem proving systems. However, because of their small state space, it is a simple problem for Model Checkers. Thus, many theorem proving systems integrate Model Checkers as decision procedure, e.g., PVS [ORS97, ORSSC98, Rus00a]. The theorem prover is used to abstract the data paths, which makes Model Checking feasible.

Besides the lack of application specific decision procedures, the obstacle for the verification of large designs is the size of the decision problems. In general, the decision procedures of most theorem provers are not optimized for large decision problems. As an example, the correctness of many hardware designs, such as arithmetic units, is a pure propositional logic problem. Current decision procedures are able to handle even complex designs with thousands of gates completely automated. However, such a problem is just too big for the decision procedures of theorem provers such as ACL2 or PVS and requires a manual proof.

1.2 Related Work

Boulton et al. [BGG+92] formalize the semantics of several hardware description languages in HOL. They provide support of viewing problems in the original design language by translating back HOL to the original HDL. The specification can be done in higher order logics. They do not discuss application specific decision procedures for the verification problem. In [Bou97], the authors provide a framework to automate the integration of an application specific language into a formal reasoning tool. Support for generating parsers and pretty-printers and internal representations is provided.

As described above, PVS integrates a Model Checker in order to decide smaller sub problems automatically. It also provides means to abstract data types in order to make problems finite or small enough. It does not provide support to input or output application specific languages. ICS [FORS01], which will be integrated in future versions of PVS, provides automated support for bit vector decision problems. However, bit-vector multiplication and shifting by variable distance is not supported.

While the PVS language allows concise specifications, the type checking problem becomes undecidable. Our experiments with PVS show that the speed of the type check becomes critical for large projects; the verification of the actual proof is faster than the type check. The type check has to be performed again after any change to the theory, so it is part of the interaction.

The Stanford Pascal Verifier [Luc79] takes Pascal as input language and contains decision procedures specialized for the task. However, it does not contain further reasoning capabilities. Non-trivial verification conditions have to be verified by means of an external, generic theorem prover. The extended static checker [DLNS98] takes Java programs as input and also contains decision procedures that are specialized for this task; however, the tool focuses on specific, simple properties and does not aim at a full proof of correctness.

Other application specific verification tools include AX / SPIN and the SLAM project for ANSI-C. However, these tools lack a full higher-order logic theorem prover. The aim of the LOOP project is to formally verify Java programs by conversion to the language of a variety of theorem provers. However, the proofs are done using the language of the theorem prover. The specifications are done in a special language.

The combination of a functional programming language and theorem provers is much more natural than the combination of sequential programming languages and theorem provers. Thus, there are theorem provers that accept functional programming languages as input. ACL2 is both used as executable programming language and as specification language for the theorem prover. Another example is OCaml and Nuprl. A significant advantage is that this allows the theorem prover to argue about its own code.

When designing application specific theorem proving systems, many components can be re-used. The SyMP framework [Ber01, Ber02] allows the integration of application specific proof systems. The user interface and the proof manager are shared among all proof systems. Every proof system can provide an application specific input/output language. As examples, SyMP provides languages for security protocols and a language that is used for the verification of cache coherence protocols. In one of the proof systems, the Model Checker SMV is tightly integrated by extending the Gentzen sequent with an explicit model and temporal operators. The interaction with the user is done using the application specific language only.

However, the framework does not integrate the application specific languages by means of an internal higher-order logic representation. The disadvantage of this approach is that it prohibits the sharing of proof commands and decision procedures. No common specification language is provided.

1.3 Contribution

We provide theorem proving support for application specific design languages by extending the SyMP framework by a common, higher-order logic internal representation. All design languages are converted into this representation. Design languages currently implemented are ANSI-C, Verilog, and the SMV language. In addition to the design languages, we import the language of the PVS theorem prover, with minor restrictions. This language is used as common specification language for all design languages. In comparison with PVS, we improve the speed of the type check of PVS input files significantly. The user is able to pick the language for the interaction. This can be any design language or the specification language. The language can be switched while proving a theorem.

Besides the usual set of higher-order logic proof rules, we implement decision procedures that allow verifying larger sub-problems. We implement a SAT based decision procedure for bit vector arithmetic including multiplication. We implement a equality logic decision procedure that automatically applies user provided lemmas in the style of rewriting rules. In contrast to prior approaches, the algorithm is SAT based and allows large decision problems and, on the propositional level, arbitrary rules.

1.4 Outline

In section 2, we describe the overall framework of our approach. In section 3, we describe the hardware description language support, the ANSI-C support, and details of the PVS language module. In section 4, we describe the decision procedures.

2 Framework

Figure 1 shows an overview of the tool and the framework. Our tool is integrated into the SyMP (Symbolic Model Prover) framework, which was developed by Sergey Berezin [Ber02].

SyMP provides the proof manager, which manages the proof trees and controls the interactive proof construction. The proof manager interfaces to the user interface and the proof systems. SyMP comes with an Emacs user interface, which is much like

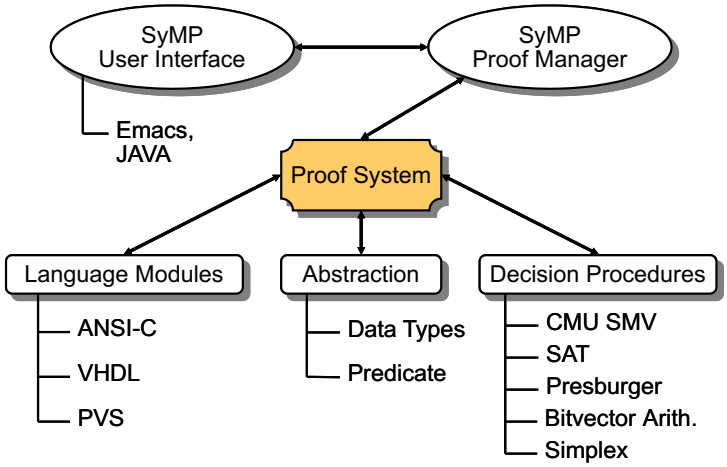


Figure 1: Tool Overview

the user interface of PVS. Flavio Lerda has implemented a new, graphical user interface in Java, which allows ”push-button” proof construction.

While the user interface and the proof manager are language and application independent, the proof systems of SyMP are application specific. They provide language support by implementing the parser and pretty-printer. We extend the SyMP framework by implementing one proof system that supports multiple, application specific languages by means of language modules.

A language module translates a given input language into a common, higher-order logic internal representation. It also translates the internal representation back into the application specific language for displaying proof obligations in the original language. This way, the engineer uses the design language during the whole design and verification process. The theorem prover integrates well in the design process.

The proof system also contains a common set of proof rules and decision procedures, which are shared. Thus, the proof rules and decision procedures are implemented only once and are used for all languages.

This concept also allows comparing designs and specifications that are equivalent but made in different languages. For example, one can compare ANSI-C designs to hardware designs given in Verilog or PVS language.

Internal Representation The internal representation is a tree-like data structure allowing arbitrary higher-order logic. After type check, every node in the tree is annotated with a type. Of all languages we implement, PVS has the most expressive type system. We therefore use the same type system as implemented by PVS: A type consists of a base type and a predicate. The base type is a basic type, such as the number type or an enumerative type, or combinations of basic types (tuples, records, ...). The predicate allows generating sub-types from basic types.

3 Language Modules

3.1 Hardware Description Languages

We implement conversion into and from the internal representation for Verilog [TM91] and the language of SMV, an automated Model Checker. This allows reading hardware designs directly into the tool. The formalization of the languages in higher-order logics is simple. In case of Verilog, we assume that the design is given as clocked circuit with only one clock. Besides the registers, the circuit has to be completely combinatorial. We therefore import a record type, which represents the registers, and a transition function. We do not support any non-synthesizable language features. We do not generate theorems from Verilog files.

In case of the SMV language, we import a record type, which represents the defined variables and their types, the transition relation, and an initial state predicate. We also import any given specification, which may contain temporal operators, as a theorem.

3.2 ANSI-C

3.2.1 Formalization

The semantics of ANSI-C are defined in the 1999 ISO/IEC C Standard ("C99") [Int99], henceforth referred to as "the standard". They provide many options to the implementation of the compiler. Examples include the bit codings of the number types, padding of structures, signedness of several data types, and the sub-expression evaluation order. This has both advantages and disadvantages. The advantage is that ANSI-C can be used in many environments, and easily and efficiently adapts to many different architectures.

The disadvantage is that the behavioral semantics of ANSI-C programs are quite often not well defined or depend on the actual architecture. This proposes a challenge for formal verification.

One of the main parts of the project therefore is the formalization of the semantics of ANSI-C. For this task, we extend Hoare's logic with concepts required for ANSI-C such as functions with side effects, pointers and so on. We offer two ways to handle the options allowed by the standard:

1. **Fix it.** One option is to allow the user to pick a particular option. For example, we require the user to define the number of bits of the ANSI-C arithmetic data types such `int` and `char`.

2. **Verify them all.** Committing to one option is not useful in several cases. One example is the evaluation order of sub-expressions. For example, in the expression

   ```
   a = f(x) + g(x);
   ```

   the order in which the functions `f` and `g` are called is not defined by the standard. The compiler may call them in any order. In this situation, we require that the behavior of the circuit shall match the behavior of the program regardless of the order in which `f` and `g` are executed, i.e., we allow the program to pick an ordering nondeterministically.

Another example is the semantics of several operators, such as bit-wise shifting. These operands only return well-defined results on signed integers in case the operands are not negative. We return an arbitrary result, which is chosen nondeterministically, in case an operand is negative.

We formalize the semantics of the ANSI-C standard using an extended but traditional Hoare logic, which is easily embedded within higher-order logic. Hoare's logic allows tools that are intuitive to use for software engineers since technical details such as program counters (PC) are hidden by the logic.

Using the traditional Hoare axioms, the proof of correctness of the program is constructed by beginning at the end of the program. Using the traditional axioms, we conclude rules that allow to construct the proof in program execution order, which is more intuitive for engineers. This is the "forward-version" of the assignment axiom:

$$\frac{\{\exists a' : p[a/a'] \wedge a = t[a/a']\} S \{q\}}{\{p\} \quad a = t; \quad S \quad \{q\}}$$

After skolemization of the existential quantifier, this rule technically is just variable renaming, which can be performed automatically and efficiently.

There is a similar rule for arrays, which includes constraints for the array bounds in order to assert that the semantics of the array access are well-defined. Note that most security flaws in software systems written in ANSI-C are caused by out-of-bounds pointers.

$$\frac{\{\exists a' : p[a/a'] \wedge a[j] = \left\{ \begin{array}{lcl} t[a/a'] & : & j = i \\ a'[j] & : & \text{otherw.} \end{array} \right. \} S \{q\}, i \geq 0 \wedge i < size(a)}{\{p\} \quad a[i] = t; \quad S \quad \{q\}}$$

There are similar rules for all other ANSI-C constructs, such as `for`, `switch`, function calls and so on.

3.2.2 Pointers and Dynamic Memory

ANSI-C programs make heavy use of pointers. They are used to implement arrays and pass-by-reference for function call arguments.

Let $\mathcal{V}$ be the set of variables (before renaming), and let $\mathcal{F}$ be the set of functions. We assume that both are disjunct. Furthermore, the special symbol `NULL` must not be an element of either set.

Pointers are modeled as tuple with two components (v, o). The first component v is an element of the set of variables (before renaming), the set of functions, or the special value `NULL`. Let $p.v$ denote the first component of pointer p, $p.o$ the second.

$$p = (v, o)$$

$$p.v \in \mathcal{V} \cup \mathcal{F} \cup \{\texttt{NULL}\}$$

If $p.v$ is an element of the set of variables, $p.o$ is used to denote the offset within that variable. Otherwise, $p.o$ is not used. Pointers are dereferenced using the following rule: if $p.v$ is equal to variable $a \in \mathcal{V}$ and $p.o$ is within the bounds of the variable, $*p$ is equal to a. Variables that are not of an array type have size 1.

$$p.v = a \wedge p.o \geq 0 \wedge p.o < size(a) \implies *p = a[p.o]$$

Example: Consider the following code fragment:

```
int a[4], b, *p;
if(x) p=&a[2]; else p=&b;
```

After the execution of the second line, the pointer p may have two values, depending on x: $(a, 2)$ or $(b, 0)$.

In many cases the constant propagation provides fixed values for both components. In this case, a read or write access to a dereferenced pointer is handled just as a variable read or write access. However, in practice there are multiple possibilities if assignments are made to pointers that depend on input variables. In this case, a case split that considers all possible values of the two components is performed. This can yield significant blowup.

We optionally add subgoals that assert that p points to a valid variable and that the offset is within the bounds of the variable (left hand side of implication above). This allows checking whether exceptions can occur.

Pointer Arithmetic Pointer arithmetic on p is performed by bit vector arithmetic on $p.o$:

$$p + i \quad = \quad (p.v, p.o + i)$$

The difference of two pointers is only defined if the pointers point to the same variable $a \in \mathcal{V}$. We add a subgoal $p.v = q.v$.

$$p.v = a \wedge q.v = a \implies p - q = p.o - q.o$$

Dynamic Memory Allocation We verify programs that make use of dynamic memory allocation, e.g., for dynamically sized arrays or data structures such as lists or graphs. This is easily realized by replacing every call to `malloc` or `calloc` by the address of a new variable. For this, we assume that the type of the new variable is given by ether an explicit or implicit type cast to a pointer that points to a variable of type t. In case of `malloc`, let x be a new variable of an array type with elements of type t and size $s/$`sizeof(`t`)`. We assert that s is an integer multiple of `sizeof(`t`)`.

$$\texttt{(}t\texttt{ *) malloc(}s\texttt{)} \longrightarrow \texttt{\&}x$$

In case of `calloc`, let n denote the number of elements to be allocated and s denote the size of each element. We add a subgoal that asserts that s matches `sizeof(`t`)`. Let x be a new variable of an array type with elements of type t and size n.

$$\texttt{(}t\texttt{ *) calloc(}n\texttt{, }s\texttt{)} \longrightarrow \texttt{\&}x$$

```
{-1}  output == 0
[-2]  factor2 == in2
[-3]  factor1 == in1
--------------------------
{
  while(factor1 != 0)
  {
    if(factor1 & 1) output = output + factor2;

    factor1 = factor1 >> 1;
    factor2 = factor2 << 1;
    assert(output == in1 * in2 - factor1 * factor2);
  }
}
--------------------------
[1]  output == in1 * in2
```

Figure 2: Combination of Hoare Triple and Gentzen sequent

3.2.3 Proof Construction for ANSI-C Programs

Using Hoare's axioms, and given that loops have an upper run-time bound, the ANSI-C program can be transformed into a Boolean propositional formula completely automated by a proof strategy. Also given that the specification written in bit vector logic, the verification problem is within bit vector logic and therefore decideable. We also support arbitrary higher-order logic properties given as PVS specification. In this case, the proof has to be constructed manually.

The user interaction is done using a variant of the Gentzen sequent: In the Gentzen sequent, the verification problem is displayed as implication as follows:

$$\bigwedge_i a_i \implies \bigvee_j c_j$$

We combine the Gentzen sequent and the Hoare triple as follows: the precondition is displayed as conjunction, and the post-condition is displayed as disjunction:

$$\{\bigwedge_i a_i\} \quad \texttt{code} \quad \{\bigvee_j c_j\}$$

This allows using a consistent set of proof commands, such as `copy` and `delete` for the manipulation of sequents for both pure Gentzen sequents and Hoare triples. See figure 2 for an example.

3.3 PVS Language as Specification Language

The language of the theorem proving system PVS was chosen as common specification language. It is well-established in contrast to new approaches such as SAL [BGL+00]. As described above, the type check is time critical on large theories.

Restrictions We support the full set of language features of the PVS language with the following restrictions: For efficient parsing, we require a semicolon after each definition. This is optional in the original PVS language. We do not allow overloading the keyword `IF` (the parser is taken from [BBJ+02]). We do not resolve overloading ambiguities using the type of the expression, but rather only the type of arguments. Thus, the type of an expression only depends on the expression itself, never on its context.

We are using a state of the art microprocessor design as benchmark [BJ01, BBJ+02]. The microprocessor contains an out-of-order scheduler and a floating point unit. The design, specification and proof are done using PVS. Experiments show that the type check is time critical since it is part of the interaction. For large context, the type check using PVS takes up to thirty minutes. Our implementation takes less than four seconds for the same input. All times are obtained on a dual AMD Athlon with 1.5 GHZ and 3 GB of RAM. This result shows that our implementation is fast enough even for large designs that are too big for generic theorem provers.

4 Decision Procedures

4.1 Overview

Traditional theorem proving puts to much burden, i.e., manual work, on the verification engineer. By adding strong decision procedures as rule of inference, we can provide a very high degree of automation. Typically, the verification engineer splits the whole design into smaller parts or modules, which can then be verified automatically by automated decision procedures. The decision procedures that come with generic theorem provers are usually too slow for big decision problems. We therefore implement efficient decision procedures for subgoals that are large. Furthermore, most are optimized for specific applications.

The following decision procedures are implemented:

- For simplification, we have implemented a simple constant propagation and canonization algorithm. This is similar to the `simplify` rule in PVS.
- We implemented CMU SMV as decision procedure for smaller problems that include a model. Experiments showed that SMV is particularly useful for the verification of liveness properties of microprocessor control circuits.
- We implemented a strong SAT based decision procedure for bit vector decision problems. In the context of this project, the bit vector decision problems arise from the ANSI-C programs. We employ Chaff [MMZ$^+$01] as SAT checker. This decision procedure is described in more detail below.
- We integrated the Omega library [PW94] as decision procedure for Pressburger arithmetic.
- We implemented a variant of the Simplex algorithm for linear arithmetic on rational numbers.
- We implemented a rule based decision procedure using a SAT checker. This decision procedure is described in more detail below.

4.2 Proving Bit-vector Equations using SAT

Using Hoare style inference rules, we reduce the problem to Boolean predicates on the state of the program. It is therefore left to verify these equations. In case of ANSI-C programs and HDL specifications, these equations are bit-vector equations. We implemented a decision procedure for such equations that translates these equations into CNF. The translation is done the same way as done by BMC, i.e., by adding new variables. There is a set of alternative algorithms for bit vector decision problems [CMR97, MR98].

We implemented numerous operators, including shifting with variable distance and arithmetic operators including multiplication. We made experiments using the SAT checker *Chaff* [MMZ$^+$01].

The results are very promising unless nonlinear arithmetic is involved. Even large formulae are asserted within seconds, given that they do not contain nonlinear arithmetic. We experienced an exponential blowup on equations that include nonlinear arithmetic, as in the following example:

$$a * b \stackrel{!}{=} (a << 1) * (b >> 1) + \begin{cases} a & : \quad b\&1 \neq 0 \\ 0 & : \quad \text{otherwise} \end{cases}$$

This equation is the induction step of an ANSI-C program that does multiplication using left and right shifting (a has to be large enough to prevent an overflow during the left shift operation).

4.3 Proving Equations using Rules and SAT

Because of the limits of the approach described above, we also implemented a rule based decision procedure for bit vector equations. We convert the formula into a Boolean propositional formula by reducing all properties to equalities. These equalities are mapped to Boolean variables. The claim and the equality based rules are passed to the SAT checker Chaff. This allows even infinite data types such as natural numbers.

We have implemented rules for most common ANSI-C operators such as addition, multiplication, relational operators etc. For example, if the sequent contains an expression such as `a+b`, we add a commutativity constraint

$$a + b = b + a$$

In case of arbitrary functions f we do not have any information about, we just make them uninterpreted using congruence closure by adding

$$a = b \Longrightarrow f(a) = f(b)$$

as constraint. The algorithm queries all theorems and lemmas that have been verified so far and adds all appropriate theorems as constraint.

This certainly provides no completeness, but is able to decide many equations that arise from practical examples completely automated. The algorithm is very fast, and the generated CNF formulae typically only have a few thousand variables and are verified in less than a second by Chaff.

5 Conclusion and Future Work

We illustrated how to combine several completely different application specific design languages within one common higher-order logic theorem proving framework. We describe a tool implementing ANSI-C, Verilog, SMV, and PVS language.

Future work includes the addition of more design languages such as VHDL and object oriented languages like Java and C++, and more application specific decision procedures. In particular, for modular reasoning about languages with pointers or objects a decision procedure for a logic like separation logic is desirable.

References

[BBJ+02] Christoph Berg, Sven Beyer, Christian Jacobi, Daniel Kröning, and Dirk Leinenbach. Formal verification of the VAMP microprocessor (project status). In Witold Charatonik and Harald Ganzinger, editors, *Symposium on the Effectiveness of Logic in Computer Science (ELICS02),*, pages 31–36, 2002. Technical Report MPI-I-2002-2-007, Max-Planck-Institut für Informatik, Saarbruecken, Germany.

[Ber01] Sergey Berezin. The SyMP tool. `http://www.cs.cmu.edu/~modelcheck/symp.html`, 2001.

[Ber02] Sergey Berezin. *Model Checking and Theorem Proving: a Unified Framework*. PhD thesis, Carnegie Mellon University, January 2002.

[BGG+92] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, volume A-10 of *IFIP Transactions*, pages 129–156, Nijmegen, The Netherlands, June 1992. North-Holland/Elsevier.

[BGL+00] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Mu noz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, jun 2000. NASA Langley Research Center.

[BJ01] Christoph Berg and Christian Jacobi. Formal verification of the VAMP floating point unit. In *Proc. 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 2144 of *LNCS*, pages 325–339. Springer, 2001.

[Bou97] R. J. Boulton. A tool to support formal reasoning about computer languages. In E. Brinksma, editor, *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 81–95, Enschede, The Netherlands, April 1997. Springer.

[BT00] P. Bertoli and P. Traverso. Design verification of a safety-critical embedded verifier. In M. Kaufmann, P. Manolios, and J Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, 2000.

[CGM86] A. Camilleri, M. Gordon, and T. Melham. Hardware verification using higher order logic. In *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 41–66. North-Holland, 1986.

[CMR97] David Cyrluk, Oliver Möller, and Harald Rueß. An efficient decision procedure for the theory of fixed-sized bit-vectors. In Orna Grumberg, editor, *9th International Conference on Computer-Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 60–71. Springer-Verlag, 1997.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report 159, Compaq SRC Research Report, 130 Lytton Ave., Palo Alto, 1998.

[FORS01] Jean-Christophe Filliâtre, Sam Owre, Harald Rueß, and N. Shankar. Ics: Integrated canonizer and solver. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th Conference on Computer-Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[Gor85] Michael J. C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In George J. Milne and P. A. Subrahmanyam, editors, *Proceedings of the 1985 Edinburgh Workshop on VLSI Design: Formal Aspects of VLSI Design*, pages 153–177, Edinburgh, Scotland, 1985. North Holland.

[Har99] John Harrison. A machine-checked theory of floating point arithmetic. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*, pages 113–130, Nice, France, September 1999. Springer-Verlag.

[Int99] International Organization for Standardization. *ISO/IEC 9899:1999: Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, 1999.

[KM96] Matt Kaufmann and J. S. Moore. ACL2: An industrial strength version of nqthm. In *Proc. of the Eleventh Annual Conference on Computer Assurance*, pages 23–34. IEEE Computer Society Press, 1996.

[Luc79] D. Luckham. Stanford Pascal verifier user manual. Technical Report STAN-CS-79-731, Stanford University Computer Science Department, March 1979.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.

[MR98] M. Oliver Möller and Harald Rueß. Solving bit-vector equations. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Formal methods in computer aided design: Second International Conference, FMCAD'98*, number 1522 in LNCS, pages 36–48. Springer-Verlag, 1998.

[ORS97] Sam Owre, John Rushby, and N. Shankar. Integration in PVS: tables, types, and model checking. In Ed Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems TACAS '97*, number 1217 in Lecture Notes in Computer Science, pages 366–383, Enschede, The Netherlands, April 1997. Springer-Verlag.

[ORSSC98] Sam Owre, John Rushby, N. Shankar, and David Stringer-Calvert. PVS: an experience report. In Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullman, editors, *Applied Formal Methods—FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 338–345, Boppard, Germany, October 1998. Springer-Verlag.

[Pfe00] Holger Pfeifer. Formal verification of the TTP group membership algorithm. In Tommaso Bolognesi and Diego Latella, editors, *Formal Methods for Distributed System Development Proceedings of FORTE XIII / PSTV XX 2000*, pages 3–18, Pisa, Italy, October 2000. Kluwer Academic Publishers.

[PSvH99] Holger Pfeifer, Detlef Schwier, and Friedrich W. von Henke. Formal Verification for Time-Triggered Clock Synchronization. In Charles B. Weinstock and John Rushby (eds.), editors, *Dependable Computing for Critical Applications 7*, volume 12 of *Dependable Computing and Fault-Tolerant Systems*, pages 207–226. IEEE Computer Society, January 1999.

[PW94] William Pugh and David Wonnacott. Static analysis of upper and lower bounds on dependences and parallelism. *ACM Transactions on Programming Languages and Systems*, 16(4):1248–1278, July 1994.

[Rus97] John Rushby. Subtypes for specifications. In Mehdi Jazayeri and Helmut Schauer, editors, *Software Engineering–ESEC/FSE '97: Sixth European Software Engineering Conference and Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1301 of *Lecture Notes in Computer Science*, pages 4–19, Zurich, Switzerland, September 1997. Springer-Verlag.

[Rus00a] John Rushby. Theorem proving for verification. In Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Dermot Ryan, editors, *Modelling and Verification of Parallel Processes: MOVEP 2000*, number 2067 in Lecture Notes in Computer Science, pages 39–57, Nantes, France, June 2000. springer Verlag.

[Rus00b] David Russinoff. A case study in formal verification of register-transfer logic with ACL2: The floating point adder of the AMD Athlon processor. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference (FMCAD 2000)*, volume 1954 of *Lecture Notes in Computer Science*. Springer Verlag, 2000.

[TM91] Donald E. Thomas and Philip Moorby. *The Verilog Hardware Description Language*. Kluwer, Boston;Dordrecht;London, 1991.

[V. 95] V. Pratt. Anatomy of the Pentium Bug. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development*, number 915 in Lecture Notes in Computer Science, pages 97–107. Springer Verlag, 1995.

The TRUST protocol analyser
Automatic and efficient verification of cryptographic protocols

Vincent Vanackère
Laboratoire d'Informatique Fondamentale de Marseille
Université de Provence,
39 rue Joliot-Curie, 13453, Marseille, FRANCE
vanackere@cmi.univ-mrs.fr

June 2002

Abstract

This paper presents TRUST, a verifier for cryptographic protocols. In our framework, a protocol is modeled as a finite number of processes interacting with an hostile environment; the security properties expected from the protocol are specified by inserting logical assertions on the environment knowledge in the processes.

Our analyser relies on an exact symbolic reduction method, combined with several techniques aiming to reduce the number of interleavings that have to be considered. We argue that our verifier is able to perform a full analysis on up to 3 parallel (interleaved) sessions of most protocols. Moreover, authentication and secrecy properties are specified in a very natural way, and whenever an error is found an attack against the protocol is given by our tool.

Keywords: cryptographic protocols, symbolic verification, state explosion problem

1 Introduction

The aim of this paper is to present TRUST, a verifier for cryptographic protocols relying on a symbolic reduction method introduced in [AL00] and further developed in [ALV01]. Although the symbolic reduction system allows us in theory to perform an exact analysis of an otherwise infinitely branching system, we face the same problem as in most model-checking tools: as the number of parallel threads goes up, the number of possible interleavings make the verification task harder - if not impossible - because of the state-space explosion problem. It should be noted that the verification problem we are discussing here was shown to be NP-complete [ALV01, RT01].

Our primary goals while developing this implementation were efficiency and ease of use. Most notably we use an *eager reduction* procedure in order to minimize the number of interleavings that have to be considered. Together with that, we have explored - and used - several symmetry and partial order reductions techniques. The end result is that our tool is able to handle up to 2 or even 3 parallel sessions of most protocols. We found by experience that inserting assertions within a protocol is a very natural way to specify security properties and is a good way to very quickly find a flaw. Our verifier handles nonces, symmetric and asymmetric keys; assertions consist of arbitrary boolean combinations of tests on equality, secrecy and authentication.

2 Theoretical background

Our formal model is presented in details in [ALV01], therefore we will only give a short presentation here.

We use the common Dolev-Yao model [DY83], where the network is under full control of an adversary that can analyse all messages exchanged and synthetize new ones. We work under the perfect encryption assumption, thus messages can be viewed as terms in a free algebra. We distinguish between basic names (agent's names, nonces, keys,...) and composed messages (pairs $< \_, \_ >$ and encrypted terms $E(\_, \_)$), with the restriction that only basic names may be used as encryption keys. The set of names is denoted by $\mathcal{N}$ and the full set of messages by $\mathcal{M}$.

2.1 Analysis and synthesis

The intruder capabilities are formally defined from two operators doing the analysis and synthesis on a set of messages.

We assume a (computable) relation $\mathcal{D} \subseteq \mathcal{N} \times \mathcal{N}$ with the following interpretation:

$$(C, C') \in \mathcal{D} \text{ iff messages encrypted with } C \text{ can be decrypted with } C'.$$

We define $Inv(C) = \{C' \mid (C, C') \in \mathcal{D}\}$. Further hypotheses, on the properties of $\mathcal{D}$ allow to model hashing, symmetric, and public keys. In particular: (i) for a *hashing* key C, $Inv(C) = \emptyset$, (ii) for a *symmetric* key C, $Inv(C) = \{C\}$, and (iii) for a *public* key C there is another key C' such that $Inv(C) = \{C'\}$ and $Inv(C') = \{C\}$.

Given a set of terms T we can now define the S (synthesis) and A (analysis) operators as follows :

- $S(T)$ is the least set that contains T and such that:

$$\begin{array}{ll} t_1, t_2 \in S(T) & \Rightarrow \langle t_1, t_2 \rangle \in S(T) \\ t_1 \in S(T), t_2 \in T \cap \mathcal{N} & \Rightarrow E(t_1, t_2) \in S(T) \ . \end{array}$$

- $A(T)$ is the least set that contains T and such that:

$$\begin{array}{ll} \langle t_1, t_2 \rangle \in A(T) & \Rightarrow t_i \in A(T),\ i = 1, 2 \\ E(t_1, t_2) \in A(T), A(T) \cap Inv(t_2) \neq \emptyset & \Rightarrow t_1 \in A(T) \ . \end{array}$$

As an example, if $T = \left\{E(\langle A, B\rangle, K), K^{-1}\right\}$, then $A(T) = T \cup \{A, B, \langle A, B\rangle\}$ and *e.g.* $E(A, K^{-1}) \in S(A(T))$. Using these definitions, the set of messages that an adversary can derive from T is $S(A(T))$; a trivial - but quite important - remark is that this set will be infinite as soon as T is not empty.

2.2 Processes and configurations : semantics

In our framework, a protocol is modelled as a finite number of processes interacting with an environment. As our process syntax includes the parallel composition - commutative and associative - of two processes, we can define a configuration as a couple (P, T) where P is a process and T a set of terms representing the current adversary knowledge, that is the initial knowledge augmented with all messages emitted by the participants of the protocol so far.

Figure 1 gives the semantic rules as a reduction system on configurations. Informally, a process can either:

(!) Write a message : the term is simply added to the environment knowledge.

(?) Read some message from the environment : this can be any message the adversary is able to build from its current knowledge.

$$
\begin{array}{lll}
(!) & (\mathsf{write}\, t.P \mid P', T) & \rightarrow (P \mid P', T \cup \{t\}) \text{if } t \in \mathcal{M} \\
(?) & (\mathsf{read}\, x.P \mid P', T) & \rightarrow ([t/x]P \mid P', T) \text{if } t \in S(A(T)) \\
(\mathsf{d}) & (x \leftarrow \mathsf{dec}(E(t, C), C').P \mid P', T) & \rightarrow ([t/x]P \mid P', T) \text{if } C' \in Inv(C), t \in \mathcal{M} \\
(\mathsf{pl}) & (x \leftarrow \mathsf{proj}_l(\langle t, t' \rangle).P \mid P', T) & \rightarrow ([t/x]P \mid P', T) \text{if } t, t' \in \mathcal{M} \\
(\mathsf{a}) & (\mathsf{assert}(\varphi).P \mid P', T) & \rightarrow \begin{cases} (P \mid P', T) & \text{if } \models_T \varphi \\ \mathsf{err} & \text{if } \not\models_T \varphi \end{cases} \\
(\mathsf{m}_1) & ([t = t]P_1, P_2 \mid P', T) & \rightarrow (P_1 \mid P', T) \text{if } t \in \mathcal{M} \\
(\mathsf{m}_2) & ([t = t']P_1, P_2 \mid P', T) & \rightarrow (P_2 \mid P', T) \text{if } t \neq t', t, t' \in \mathcal{M}
\end{array}
$$

Figure 1: Reduction on configurations

(d) Decrypt some (encrypted) term with a corresponding inverse key.

(pl) Perform some unpairing (the symmetric rule (pr) is not written).

(m_i) Test for equality/inequality of two messages.

(a) Check if some assertion φ holds.

Missing from the figure is the terminated process, denoted by 0, as well as the syntax of the assertion language, that will be presented in the next section. err denotes a special configuration that can only be reached from a false assertion.

In our model, *a correct protocol is a protocol that cannot reach the* err *configuration* - or, put in other words, a protocol such that all assertions reachable from the initial configuration of the system hold.

2.3 Specifying security properties through assertions

The full assertion language we consider is the following:

$$\varphi ::= \quad true \mid false \mid t = t' \mid t \neq t' \mid \mathsf{known}(t) \mid \mathsf{secret}(t) \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2$$

This is equivalent to saying that we consider arbitrary boolean combinations of atomic formulas checking the equality of two messages $t = t'$ and the secrecy of a message $\mathsf{secret}(t)$ with respect to the current knowledge of the adversary. As shown in [ALV01], this language allows to easily express authentication properties such as aliveness and agreement ([Low97]).

We take as a short example the following 3 message version of the Needham-Schroeder Public Key protocol:

$$
\begin{array}{ll}
A \rightarrow B: & \{na, A\}_{Pub(B)} \\
B \rightarrow A: & \{na, nb\}_{Pub(A)} \\
A \rightarrow B: & \{nb\}_{Pub(B)}
\end{array}
$$

In our framework, the protocol can be modeled as follows (the open variables are to be instantiated by the process and peer identities before the actual symbolic reduction):

$$
\begin{array}{ll}
Init(myid, resp): & \text{fresh } na. \\
& \mathsf{write}\ E(\langle na, myid \rangle, Pub(resp)). \\
& \mathsf{read}\ e.\ \langle na', nb \rangle \leftarrow \mathsf{dec}(e, Priv(myid)).\ [na' = na]. \\
& \mathsf{write}\ E(nb, Pub(resp)). \\
& \mathsf{assert}(\mathsf{secret}(nb) \wedge \mathrm{auth}(resp, myid, na, nb)).\ 0 \\
\\
Resp(myid, init): & \mathsf{read}\ e.\ \langle na, a \rangle \leftarrow \mathsf{dec}(e, Priv(myid)).\ [a = init]. \\
& \text{fresh } nb. \\
& \mathsf{write}_{\mathsf{auth}(myid, init, na, nb)}\ E(\langle na, nb \rangle, Pub(init)). \\
& \mathsf{read}\ e'.\ [e' = E(nb, Pub(myid))]. \\
& 0
\end{array}
$$

The instruction "$\mathsf{write}_{\mathsf{auth}(msg)}t$" is some syntactic sugar to be replaced by "$\mathsf{write}\ \langle E(msg, K_{\mathsf{auth}}), t\rangle$", whereas the assertion "$\mathrm{auth}(msg)$" is a shortcut for "$\mathsf{known}(E(msg, K_{\mathsf{auth}}))$". These notations are actually supported by our tool and their usage reveals itself quite convenient in practice. In our example, the initiator specifies that at the end of its run of the protocol, the nonce nb must be secret and expects an agreement with some responder on the nonces na and nb.

2.4 Symbolic reduction

The main difficulty in the verification task is the fact that the input rule (?) is infinitely branching as soon as the environment is not empty. In [AL00, ALV01] it was shown that it is possible to solve this problem by using a symbolic reduction system that stores the constraints in a symbolic shape during the execution. As an example, the input rule $(\mathsf{read}\ x.P, T) \rightarrow ([t/x]P, T), t \in S(A(T))$ becomes $(\mathsf{read}\, x.P, T, E) \rightarrow (P, T, (E\,;\, x : T))$. The complete description of the symbolic reduction system can be found in [ALV01]. The main property we rely on is the fact that the symbolic reduction system is in lockstep with the ground one and provides a - sound and complete - decision procedure for processes specified using the full assertion language described in section 2.3.

3 Techniques for an efficient verification

Although the symbolic reduction system is satisfying from a theoretical point of view, an inherent limitation is that it does not handle iterated processes (as the general case for iterated processes is undecidable). Thus, in order to verify a protocol against replay attacks and/or parallel sessions attacks, it is quite important to handle cases where there is a finite - even if small - number of participants playing each role.

Of course, as the number of parallel threads goes up, the number of possible interleavings make the verification task harder - if not impossible - because of the state explosion problem. The main techniques that have been used/introduced in our tool are:

Depth-first search: this strategy brings here a lot of advantages, the main one being that no state needs to be explicitly saved (as all the necessary information indeed lies within the continuation of the program). As a consequence, the memory requirement of our tool is almost constant and quite low.

Carefully chosen data structures: substitutions are heavily used during the symbolic reduction process. By using a representation of terms as DAGs (directed acyclic graphs) where all variables are shared, substitutions on variables are done in $O(1)$ time. Other data structures (such as the one representing the environment knowledge) were chosen in order to allow for incremental computation whenever possible. These classical algorithmic optimizations do make a huge difference on the execution time: namely, the speed-up of our current tool w.r.t. our first prototype - mesured by the number of reductions per second - is greater than 500.

Pruning of equivalent schedulings of parallel processes: it is quite important not to explore all interleavings, but only those that have a significance. For this purpose, we have introduced an *eager reduction* technique that allows in some cases huge savings on the computation time. Aside from that, symmetry in the system is also exploited in order to further cut the state space.

We will now proceed in giving more details on our eager reduction procedure (section 3.1) and on the way we handle symmetry in the system (section 3.2). We then give a small note on other partial reduction techniques that may be applied.

3.1 Eager reduction

When verifying a system of parallel processes, only a small number of all possible interleavings need to be explored, because a lot of reduction steps are independent from each other[1]. While conceptually simple, the eager reduction procedure we introduced in our verifier has - to our knowlegde - never been described in the literature; this section is devoted to a high-level description of our method. Technical details and proofs can be found in appendix A.

In the following, we study the reduction of a configuration $(P_1 \mid \ldots \mid P_m, T)$, denoted by $(\Pi P_i, T)$. We will not allow the rewrite of $P \mid Q$ as $Q \mid P$, therefore we can define the relation $\rightarrow_x$ as a reduction on the x-th process of the parallel composition.

The eager reduction procedure relies on the fact that when considering a sequence of reductions $(\Pi P_i^{(1)}, T_1) \rightarrow \cdots \rightarrow (\Pi P_i^{(n)}, T_n)$ where $S(A(T_1)) = S(A(T_n))$ (i.e. the adversary knowledge does not increase during the reductions), then all reductions on the different processes are independent from each other. This leads to define a "big step" reduction *that amounts to reducing one process until it writes some term that was previously unknown to the environment*, thus we define the algorithm for an eager reduction as follows:

Algorithm 3.1 *Step of eager reduction of* $(\Pi P_i, T)$*:*

1. *Choose* $j \in [1, n]$.
2. $c := (\Pi P_i, T)$
3. *Choose* c' *such that* $c \rightarrow_j c'$.
4. *If* $c' \equiv (\Pi P'_i, T')$ *and* $S(A(T')) = S(A(T))$ *then* $\{$ $c := c'$ *; go to step* 3 $\}$ *else return c'*

A more formal definition, together with a proof of correctness and completeness, is given in appendix A.

From ground eager reduction to symbolic eager reduction

We stress on the fact that although the eager reduction procedure has been described and proved here only on the ground reduction system, our verifier in fact relies on the symbolic counterpart of it. The *symbolic eager reduction procedure* matches closely the ground one, the only difference being that we (symbolically) reduce a process until it reaches error or writes a term *symbolically unknown* to the environment. Completeness of the symbolic eager reduction procedure follows from the completeness of the ground reduction (but is beyond the scope of this paper).

3.2 Exploiting symmetry

When studying several parallel sessions of protocols, it is useful to define protocol *roles*, which are parametric processes. All parameters will range over a finite set of principals names $\{\mathsf{Id}_0, \ldots, \mathsf{Id}_n\}$. In our verifier, the identifier Id_0 is reserved to name a compromised participant, whereas all the names $\{\mathsf{Id}_1, \ldots, \mathsf{Id}_n\}$ are supposed to play a symmetric role in the protocol: then, we instanciate the parameters using basic injective renaming in order to generate all possible cases.

As a consequence of the completeness of eager reduction, we only need to consider "eager traces"; therefore, whenever some role is involved in a reduction to error, there is one process among those of that role that will do a step of eager ground reduction at first. Thus we can start the reduction by using only one process of each role, and *add another process of some role only after the last introduced process of the same role has performed a full step of eager reduction.* Although this may look simplistic, this allows a very important reduction in the number of states having to be explored, even when considering only 2 parallel sessions of a protocol.

[1] As a trivial example, consider two processes in parallel, one performing a decryption, and the other one an equality test: the order in which the two reduction steps are done does not affect at all the reachability of an error.

3.3 Going further...

We have also investigated some more advanced partial order techniques in order to further reduce the size of the state-space to be explored: it is namely possible, at the symbolic level, to detect that some eager reduction step was indeed independent from a previous one in the same trace. Then we can restrict the search to only explore traces that are in some (lexicographical) normal form (see [DM96]). Unfortunately, the proof of completeness for these methods become quite involved, and the gain observed in practice was not as important as expected: further investigation in this area is still needed.

4 Experimental results

TRUST was written in OCAML, and the syntax it accepts is very close to the one of the example from section 2.3 (see appendix B for a real example). This section provides some experimental results for our tool. Reassuring is the fact that our tool successfully found all known flaws on all protocol we have tried so far - even thoses the author was not yet aware of...

Benchmarks

Figure 2 gives some figures for the full analysis of some typical protocols. For each protocol, we detail the number of roles involved and give the time to do a full search depending on the number of parallel (interleaved) sessions. In that benchmark, all roles parameters ranged over a set of 3 names $\{\mathsf{Id}_0, \mathsf{Id}_1, \mathsf{Id}_2\}$, Id_0 being the name of a compromised principal whose private keys were initially known by the environment. All measures were done on a Pentium III at 733MHz, on which the tool performs more than 750.000 *basic* reductions per second. The total time spent is more or less proportional to the number of reductions done and, for instance, when verifying 3 interleaved sessions of the Needham-Schroeder protocol (with a key server), the verifier indeed performs around 88.000.000 reductions, checking more than 2.900.000 assertions.

Protocol	# roles	# sessions	time
yahalom	3	1	$< 0.01s$
yahalom	3	2	$12s$
needham-schroeder	2	3	$0.50s$
needham-schroeder	2	4	$22s$
needham-schroeder (with a key server)	3	2	$0.11s$
needham-schroeder (with a key server)	3	3	$115s$
otway-rees	3	1	$< 0.01s$
otway-rees	3	2	$1.90s$
otway-rees	3	3	$1940s$
kerberos v5	4	1	$< 0.01s$
kerberos v5	4	2	$15s$
kerberos v5	4	3	$\approx 3d$

Figure 2: Times for the analysis of various protocols

Of course, we do not avoid the state explosion problem, but nevertheless the verification task stays practical up to at least 2 or 3 parallel sessions for all the protocols we have tried so far. Moreover, an interesting feature is that the memory usage of our analyser is almost constant and quite small (around 1MByte, for all protocols tested so far).

Remark on the eager reduction: it should be noted that, depending on the protocol, experimental results have shown that our eager reduction procedure - compared to the more classical input/output interleaving semantics - gives improvements ranging from a factor of 2 to more than 100...

Finding attacks...

Here follows an example of an attack as reported by our tool. This particular one was on a (bad) variant of the Otway-Rees protocol introduced in [Pau97], whose full specification is given in appendix B:

```
0:Init(Id1,Id2) sends <N1,Id1,Id2,Crypt(<N1,Id1,Id2>,K(Id1))>

1:Resp(Id1,Id0) gets <na,Id0,Id1,e>
1:Resp(Id1,Id0) sends <na,Id0,Id1,e,N2,Crypt(<na,Id0,Id1>,K(Id1))>

2:Serv(Id0,Id1) gets <na,Id0,Id1,Crypt(<na1,Id0,Id1>,K(Id0)),N1,Crypt(<na,Id0,Id1>,K(Id1))>
2:Serv(Id0,Id1) sends <na,Crypt(<na,N3>,K(Id0)),Crypt(<N1,N3>,K(Id1))>

0:Init(Id1,Id2) gets <N1,Crypt(<N1,N3>,K(Id1))>
0:Init(Id1,Id2) sends Crypt(N4,N3)

0:Init(Id1,Id2) assert (Id2=Id0 or secret(N4))
```

A short explanation of the above example follows: Id_0 is the identity of a compromised principal whose key $K(\mathsf{Id}_0)$ is initially known by the environment, and the initiator makes the (false) assertion that either it wanted to communicate with Id_0 (in that particular trace leading to error, the initiator has identity Id_1 and wants to communicate with Id_2), or the data it sent at the last step must stay secret. This is actually not the case as clearly shown by the given attack.

It should be noted that by directly checking the secrecy of the key that the initiator gets at the end of its protocol run, we get the following - much shorter - error:

```
0:Init(Id1,Id2) sends <N1,Id1,Id2,Crypt(<N1,Id1,Id2>,K(Id1))>

0:Init(Id1,Id2) gets <N1,Crypt(<N1,Id1,Id2>,K(Id1))>
0:Init(Id1,Id2) assert (Id2=Id0 or secret(<Id1,Id2>))
```

This is a typical example of a type-flaw attack; although complex keys are not directly handled by our tool (namely, in our model, the pair <Id1,Id2> cannot be used as a valid encryption key), it is nevertheless possible to find some of those attacks with our tool.

5 Conclusion

We have presented the TRUST protocol analyser, a fully automatic verifier for cryptographic protocols. Our tool relies on a sound and complete symbolic reduction procedure: protocols are specified by the use of logical assertions on secrecy and authentication, and whenever an assertion is found to be invalid, an attack against the protocol is given. Our personal experience is that the description and specification of protocols using roles (parametric processes) and assertions is manageable even for non specialists, and is an easy way to find flaws in the protocols.

TRUST makes use of several techniques in order to alleviate the state space explosion problem. Most notably, it takes advantage of an eager reduction procedure, together with some basic symmetry reduction techniques. Experimental results show that - although the verification problem is actually NP-hard - our tool is able to handle efficiently 2 or even 3 interleaved sessions of most protocols from the literature.

As a sidenote, we believe that the idea behind our eager reduction procedure is simple and general enough to easily be adapted to other verification techniques such as those relying on tree automatas [Mon99, Gou00].

More information on our tool can be found at [Trust]. We are currently working on extending the symbolic decision method to particular cases when some processes - like a key server - can be iterated.

References

[AL00] R. Amadio and D. Lugiez. On the reachability problem in cryptographic protocols. In *Proc. CONCUR00, Springer LNCS 1877*, 2000. Also RR-INRIA 3915.

[ALV01] R. Amadio, D. Lugiez and V. Vanackère. On the symbolic reduction of processes with cryptographic functions. RR-INRIA 4147, March 2001. To appear in *Theoretical Computer Science*.

[DM96] V. Diekert and Y. Métivier. Partial commutation and traces. In G. Rozenberg and A. Salomaa, editors, Handbook of Formal Languages, Vol. 3, Beyond Words, pages 457–534. Springer-Verlag, Berlin, 1997

[DY83] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Trans. on Information Theory*, 29(2):198–208, 1983.

[Gou00] J. Goubault. A method for automatic cryptographic protocol verification. In *Proc. FMPPTA, Springer-Verlag*, 2000.

[Hui99] A. Huima. Efficient infinite-state analysis of security protocols. In *Proc. Formal methods and security protocols, FLOC Workshop, Trento*, 1999.

[Low97] G. Lowe. A hierarchy of authentication specifications. In *Proc. 10th IEEE Computer Security Foundations Workshop*, 1997.

[Mon99] D. Monniaux. Abstracting cryptographic protocols with tree automata. In *Proc. Static Analysis Symposium, Springer LNCS*, 1999.

[Pau97] L. Paulson. Proving properties of security protocols by induction. In *Proc. IEEE Computer Security Foundations Workshop*, 1997.

[RT01] M. Rusinowitch and M. Turuani Protocol insecurity with finite number of sessions is NP-complete. RR INRIA 4134, March 2001.

[Trust] http://www.cmi.univ-mrs.fr/~vvanacke/trust/

A Appendix

Eager reduction : proof of correctness and completeness

For a ground configuration $k \equiv (\Pi P_i, T)$, we define $\mu(k) = S(A(T))$. $\mu(err) = \emptyset$. We note $k(j) = P_j$.

We will first state the main lemma on which all the eager reduction process is based :

Lemma A.1 *If* $k_1 \rightarrow_i k_2 \rightarrow_j k_3$ *and* $\mu(k_1) = \mu(k_2)$, $k_3 \neq err$ *then* $\exists k_2'(k_1 \rightarrow_j k_2' \rightarrow_i k_3)$ *and* $\mu(k_2') = \mu(k_3)$.

Proof. We assume $i \neq j$ (else the result is trivial) and do a basic case analysis on the rules used to reduce P_i and P_j. All rules but (?), (a) and (!) do not depend at all from the environment nor modify it and thus the result holds whenever $\rightarrow_i \notin \{(\mathsf{a}), (?), (!)\}$ or $\rightarrow_j \notin \{(\mathsf{a}), (?), (!)\}$. On the 9 cases remaining, we can distinguish 4 relevant sub-cases by denoting $(\mathsf{r_i}) \in \{(\mathsf{a}), (?)\}$:

1. $k_1 \xrightarrow{\mathsf{r_1}}_i k_2 \xrightarrow{\mathsf{r_2}}_j k_3$
2. $k_1 \xrightarrow{!}_i k_2 \xrightarrow{!}_j k_3$
3. $k_1 \xrightarrow{\mathsf{r}}_i k_2 \xrightarrow{!}_j k_3$
4. $k_1 \xrightarrow{!}_i k_2 \xrightarrow{\mathsf{r}}_j k_3$

Cases (1), (2) and (3) are straightforward. Note that case (3) when $\mathsf{r} = (?)$ is folklore and used very broadly in the literature. Case (4) is where the eager reduction procedure will take advantage: namely we can perform the input/assert rule first and then reach k_3 after an output from the process number i, due to the fact that $\mu(k_1) = \mu(k_2)$ and that the input/assert rule does not depend on the environment T but only on the knowledge $S(A(T)) = \mu(k)$. □

Any sequence of reductions $k \rightarrow^* k'$ such that $\mu(k) = \mu(k')$ will preserve the environment knowledge : from the previous lemma, those reduction have the (nice) property that the order in which we reduce each process in the parallel composition does not matter. We will now annotate sequences of reductions to include the order in which the different processes modify the environment knowledge.

Definition A.2 *We write:*

$$
\begin{array}{lll}
(1) & k \xrightarrow{\emptyset} k' & \textit{iff} \quad k \rightarrow^* k' \textit{ and } \mu(k') = \mu(k) \\
(2) & k \xrightarrow{x} k' & \textit{iff} \quad \exists k_1 \mid k \xrightarrow{\emptyset} k_1 \rightarrow_x k' \textit{ and } \mu(k') \neq \mu(k_1) \\
(3) & k \xrightarrow{p_1,\ldots,p_n} k' & \textit{iff} \quad k \xrightarrow{p_1} k_1 \xrightarrow{p_2} \cdots \xrightarrow{p_n} k'
\end{array}
$$

Informally, $\xrightarrow{\emptyset}$ denotes any sequence of reductions that preserves the environment knowledge ; $\xrightarrow{x}$ means that the environment knowledge was not modified until the last step, where the process numbered x either performs an output of a previously unkown term, or reaches error.

$\xrightarrow{p_1,\ldots,p_n}$ is just syntactic sugar in order to shorten the notations.

Remark A.3 *If* $k \xrightarrow{*} k'$, *then there exists a sequence* $p_1, \ldots, p_n$ *such that* $k \xrightarrow{p_1,\ldots,p_n,\emptyset} k'$.

Definition A.4 ***(Eager reduction)*** *We define* $\hookrightarrow_x$, *a step of eager reduction on the process numbered* x, *as follows:*

$$k \hookrightarrow_x k' \textit{ iff } k \xrightarrow{x}_x k'$$

We will write $k \hookrightarrow_{p_1,\ldots,p_n} k'$ whenever $k \hookrightarrow_{p_1} k_1 \hookrightarrow_{p_2} \cdots \hookrightarrow_{p_n} k'$.

Informally, eager reduction on the process x means that we reduce only the process x in the configuration until either a term unknown to the environment is written, or we reach error.

Lemma A.5

1. $k \overset{x}{\rightarrow}{}^* k'$ *and* $k' \neq err$ *implies* $\exists k_1\ k \hookrightarrow_x k_1 \overset{\emptyset}{\rightarrow}{}^* k'$
2. $k \overset{x}{\rightarrow}{}^* err$ *implies* $k \hookrightarrow_x err$

PROOF.

1. $k \overset{x}{\rightarrow}{}^* k'$ implies $\exists k'' \mid k \overset{\emptyset}{\rightarrow}{}^* k'' \rightarrow_x k'$ and $\mu(k') \neq \mu(k'')$. All reductions in $k \overset{\emptyset}{\rightarrow}{}^* k''$ preserve the environment knowledge, and thus by iterating lemma A.1 we can move all reductions on x to the beginning of the sequence (details are left to the reader).

2. By the same reasoning : $k \overset{x}{\rightarrow}{}^* err$ implies $\exists k' \mid k \overset{\emptyset}{\rightarrow}{}^* k' \rightarrow_x$ err. Then we can use lemma A.1 to prove that $\exists k'' \mid k \overset{\emptyset}{\rightarrow}{}^*_x k'' \overset{\emptyset}{\rightarrow}{}^* k' \rightarrow_x err$ and such that there is no reduction on x between k'' and k'. Then $k' \rightarrow_x err$ means that $k'(x)$ is a false assertion w.r.t. $\mu(k')$ (recall that an assertion in the environment T only depends on $S(A(T))$), and as we have $k''(x) = k'(x)$ and $\mu(k'') = \mu(k)$, it implies $k'' \rightarrow_x err$. Thus $k \overset{\emptyset}{\rightarrow}{}^*_x k'' \rightarrow_x err$ and $k \overset{\emptyset}{\rightarrow}{}^*_x k'' \overset{\emptyset}{\rightarrow}{}^* k' \rightarrow_x err$

□

Theorem A.6

1. $k \overset{p_1,\ldots,p_n}{\rightarrow}{}^* k'$ *implies* $\exists k''\ k \hookrightarrow_{p_1,\ldots,p_n} k'' \overset{\emptyset}{\rightarrow}{}^* k'$
2. $k \overset{p_1,\ldots,p_n}{\rightarrow}{}^* err$ *implies* $\exists k''\ k \hookrightarrow_{p_1,\ldots,p_n} err$

PROOF.

1. Case $(n = 1)$ was done in the previous lemma. Else $k \overset{p_1,\ldots,p_n}{\rightarrow}{}^* k'$ implies $k \overset{p_1,\ldots,p_{n-1}}{\rightarrow}{}^* k'' \overset{p_n}{\rightarrow}{}^* k'$ and by induction : $\exists k_{n-1}\ k \hookrightarrow_{p_1,\ldots,p_{n-1}} k_{n-1} \overset{\emptyset}{\rightarrow}{}^* k''$. Thus $k_{n-1} \overset{\emptyset}{\rightarrow}{}^* k'' \overset{p_n}{\rightarrow}{}^* k'$ and we can write more directly: $k_{n-1} \overset{p_n}{\rightarrow}{}^* k'$. By using the previous lemma, we have $\exists k_n\ k_{n-1} \hookrightarrow_{p_n} k_n \overset{\emptyset}{\rightarrow}{}^* k'$. QED.

2. By (1) : $\exists k_n, k''\ k \hookrightarrow_{p_1,\ldots,p_{n-1}} k_n \overset{\emptyset}{\rightarrow}{}^* k'' \overset{p_n}{\rightarrow}{}^* err$. Thus $k_n \overset{p_n}{\rightarrow}{}^* err$ and $k_n \hookrightarrow_{p_n} err$.

□

Corollary A.7 *Correctness and completeness of the eager reduction method.*

PROOF. Completeness is stated in theorem A.6(2). Correctness comes trivially from $\hookrightarrow_x \subseteq \rightarrow^*$. □

B An Otway-Rees variant

The protocol we wish to verify is the following:

$$\begin{array}{ll} A \rightarrow B: & N_a, A, B, \{N_a, A, B\}_{K_a} \\ B \rightarrow S: & N_a, A, B, \{N_a, A, B\}_{K_a}, N_b, \{N_b, A, B\}_{K_b} \\ S \rightarrow B: & N_a, \{N_a, K_{ab}\}_{K_a}, \{N_b, K_{ab}\}_{K_b} \\ B \rightarrow A: & N_a, \{N_a, K_{ab}\}_{K_a} \end{array}$$

...and the raw protocol description as fed to our tool is:

```
Principals:

Init(me,him):
   [me!=him] ; [me!=Id0]
   fresh na
   write <na,me,him,E(<na,me,him>,K(me))>
   read <m,e>
   [m=na] ; <na2,kab><-decrypt(e,K(me)) ; [na2=na]
   fresh confidential
   write E(confidential,kab)
   assert( (him=Id0) or secret(confidential) )
   nil


Resp(me,him):
   [me!=him] ; [me!=Id0]
   read <na,a,b,e>
   [b=me] ; [a=him]
   fresh nb
   write <na,a,b,e,nb,E(<na,a,b>,K(me))>
   read <na2,e1,e2> [na2=na] <nb2,kab><-decrypt(e2,K(me)) [nb2=nb]
   write <na,e1>
   nil


Serv(init,resp):
   read <na,a,b,e1,nb,e2>
   [a=init][b=resp][a!=b]
   k1<-K(init)
   k2<-K(resp)
   <na1,a1,b1><-decrypt(e1,k1) ; [<na1,a1,b1>=<na,a,b>]
   <na2,a2,b2><-decrypt(e2,k2) ; [<na2,a2,b2>=<na,a,b>]
   fresh kab
   write <na,E(<na,kab>,k1),E(<nb,kab>,k2)>
   nil


Environment:

Id0 ; K(Id0)
```

Reasoning Services in the MathWeb-SB for symbolic verification of Hybrid Systems *

Christoph Benzmüller[1], Corrado Giromini[1], Andreas Nonnengart[2], and Jürgen Zimmer[1]

[1] Fachbereich Informatik
Universität des Saarlandes
Saarbrücken, Germany
{chris,corrado,jzimmer}@ags.uni-sb.de

[2] German Research Center for Artificial Intelligence (DFKI)
Saarbrücken, Germany
nonnenga@dfki.de

Abstract. Verification of non-linear hybrid systems is a challenging task. Unlike many other verification methods the deduction-based verification approach we investigate in this paper avoids approximations and operates directly on the original non-linear system specifications. This approach, however, requires the solution of non-trivial mathematical subtasks. We propose to model existing reasoning systems, such as computer algebra systems and constraint solvers, as mathematical services and to provide them in a network of mathematical tools in a way that they can reasonably support subtasks as they may occur in formal methods applications. The motivation is to make it simpler to implement and test verification approaches by out-sourcing complex but precisely identifiable mathematical subtasks for which specialised reasoners do already exists.

1 Introduction

Hybrid systems are heterogenous dynamical systems characterised by interacting continuous and discrete dynamics. The enormous presence of hybrid systems in safety critical applications, such as automated highway systems [18], air traffic management systems [22], embedded automotive controllers [3], and chemical processes [9], increasingly calls for safety guarantees. Since traditional program verification methods allow at best to approximate continuously changing environments by discrete sampling, special verification methods for hybrid systems, such as [15–17], have been developed. A frequently employed method is to model hybrid systems by hybrid automata. A hybrid automaton is a closed system with a *built-in* control structure determining when and how the system switches

* This work is supported by the EU training network CALCULEMUS (HPRN-CT-2000-00102) funded in the EU 5th framework.

between its various discrete states. Thereby the continuous behaviour in each discrete state is governed by a differential equation.

The verification method we will employ in our work is the deduction-based model checking approach for hybrid systems described in [23]. Given a specification of a hybrid system H (a hybrid automaton) and a safety property Φ the approach generates a second order formula $\lceil \Phi \rceil_H$ such that the validity of the latter guarantees that property Φ is valid for H. To support the validation of $\lceil \Phi \rceil_H$ this method eliminates second order location predicates in $\lceil \Phi \rceil_H$ one by one in order to transform $\lceil \Phi \rceil_H$ into an equivalent first order formula Ψ, if possible. With the validation of Ψ the verification approach terminates.

For the above deduction-based model checking approach we have identified the following mathematical subtasks: (1) The solution of sets of differential equations, (2) checking subsumption between sets of constraints, and (3) checking consistency of sets of constraints.

In general, solving these tasks is feasible in case of linear constraints and linear differential equations. Our aim, however, is to widen the spectrum of the approach, for instance, by allowing also non-linear constraints and differential equations. Mathematical tasks like (1) – (3) may also be relevant for other hybrid system verification approaches. For instance, [13] employed the computer algebra system MATHEMATICA to solve linear constraints. MATHEMATICA was later replaced by a more efficient implementation of a specialised constraint solving algorithm [14]. However, multiple implementations of the same kinds of mathematical services in different verification systems could and should best be avoided, especially if their realization is complex and challenging, such as in our context.

We propose to model existing reasoning systems, such as computer algebra systems and constraint solvers, as mathematical services and to provide them in a network of mathematical tools in a way that they can reasonably support subtasks as they may occur in formal methods applications. The motivation is to make it simpler to implement and test verification approaches by out-sourcing complex but precisely identifiable mathematical subtasks for which specialised reasoners do already exists. Allowedly, in case a verification approach later turns out to be successful (see for instance [14]) it may be reasonable from efficiency aspects and also from concession aspects to replace the connections to mathematical services again by fast re-implementations of the particularly needed algorithms. However, starting with the latter may dramatically slow down a quick development and implementation of new verification environments. This is particularly true in case the automation of the mathematical subtasks is already on the edge of current research, such as given in our case. This motivates our proposal to build up a network of mathematical reasoning services for formal methods. The more services will be appropriately added to such a network the more likely it will be that also other verification approaches can directly employ them (in early development stages) for the same purpose.

In this paper we illustrate the kinds of mathematical subtasks which occur in the verification approach [23] by looking at a simple non-linear hybrid system.

Our network of choice for providing the mathematical services is the mathematical software bus MathWeb-SB [20].

The outline of the paper is as follows: we first illustrate aspects of the deduction-based elimination approach for hybrid system verification and motivate different kinds of mathematical subtasks involved. We then sketch the mathematical software bus MathWeb-SB that is our mathematical network infrastructure of choice and, furthermore, discuss candidate systems suitable for tackling the identified kinds of problems. Finally we give an outlook on some first ideas and requirements on the modelling and solving of these problems in the MathWeb-SB.

2 Mathematical Subtasks in Hybrid System Verification

We briefly sketch the deduction-based model checking approach (DMC) of Nonnengart described in [23], starting from the description of a hybrid system. We also identify some of the mathematical subtasks that result from the application of this approach. Our presentation follows a very simple example of a non-linear hybrid system for which the deduction-based model checking approach results in non-linear constraints.

2.1 Structure of a Hybrid System

Hybrid systems are typically modelled as hybrid automata that are presented as finite graphs whose nodes correspond to global states (locations). The discrete dynamics, i.e. the state transitions, of the automaton is modelled by the edges of the graph. The continuous dynamics of the automaton is modelled by differential equations associated with each state.

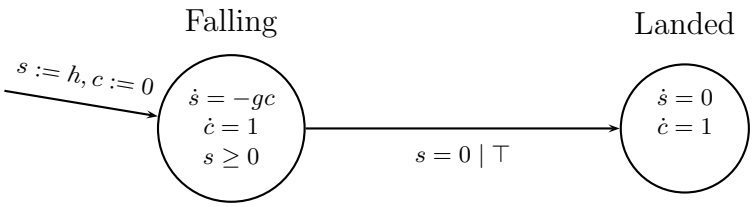


Fig. 1. A simple hybrid automaton

Fig. 1 shows a simple hybrid automaton modelling Galilei's gravity test: Given a tower of height h, we assume to stand on top of this tower and let a stone fall down. The stone falls until it reaches the bottom. Hence, we say, the stone can be in two main states: `Falling` and `Landed`. At the beginning of our

experiment the clock counter c, which mirrors the flow of time t in our system, is set to 0, and we are dropping the stone from height h. Thus, the height of the falling stone (at time t), which is represented by variable s, is initialised with h.

While the stone falls it accelerates according to the physical law of gravity:

$$\dot{s}(t) = -gt, \tag{1}$$

where g is the gravitational constant ($g = 9.81 m/sec^2$). This is represented as the invariant $\dot{s} = -gc$ in the state `Falling`. Another invariant of `Falling` is $\dot{c} = 1$ expressing that the clock c is increasing linearly. $s \geq 0$, the last invariant condition, says that the stone has not reached the ground yet. The condition for a state transition to `Landed` simply is $s = 0$.

2.2 The Deduction-Based Model Checking Approach

We now apply the DMC to our example automaton in Fig. 1. Suppose we want to know when the stone falling from the tower will reach the ground level. In terms of a *Integrator Computation Tree Logic* (ICTL) [23] formula this means to prove the property:

$$\exists\Diamond \ (Landed \wedge c \leq k)\,, \tag{2}$$

where k is a parameter to be instantiated by the proof procedure. A first task in the DMC approach is the formalisation of the reachability theory for our automaton. This theory formalises the conditions and invariants for staying in the states `Falling` and `Landed` as well as the conditions causing a transition from state `Falling` to `Landed`.

For the formalisation of this theory as a second order formula we, for instance, have to solve the differential equation (1). Taking into consideration the initialisation information we get:

$$c(t) = t, \ \ s(t) = h - \tfrac{1}{2}gt^2. \tag{3}$$

and thus

$$c(t+\delta) = c(t) + \delta, \ \ s(t+\delta) = s(t) - gc(t)\delta^2 - \tfrac{1}{2}g\delta^2 \tag{4}$$

for any change in time δ.

The complete reachability theory of the Galilei automaton is given as the following conjunctive set of second order formulas (F and L are second order variables)[1]:

$$F(s,c) \rightarrow \forall\delta.\ \left(\delta \geq 0 \wedge s' = s - gc\delta - \frac{1}{2}g\delta^2 \wedge c' = c + \delta \wedge s' \geq 0 \rightarrow F(s',c')\right)$$
$$F(s,c) \rightarrow s \geq 0$$
$$F(s,c) \rightarrow s = 0 \rightarrow L(s,c)$$
$$L(s,c) \rightarrow \forall\delta\ \ (\delta \geq 0 \wedge s' = s \wedge c' = c + \delta \rightarrow L(s',c'))$$

[1] Some of the formulas are already somewhat simplified.

$F(h,0)$ represents the initial state. Let us call this theory $\Re$. For the sake of simplicity we consider now the dual of the property (2), namely $\forall\Box(L \to c > k)$. Hence, we have to prove

$$\exists F, L \quad F(h,0) \wedge \Re \wedge \forall s, c\, L(s,c) \to \lceil L \to c > k \rceil \wedge F(s,c) \to \lceil L \to c > k \rceil$$

which simplifies to

$$\exists F, L \quad F(h,0) \wedge \Re \wedge \forall s, c\, L(s,c) \to c > k.$$

Let us first eliminate location F. We start with the fix-point computation over the state Falling.

$$\begin{array}{l} \Gamma^0(T) = T \\ \Gamma^1(T) = s \geq 0 \wedge s = 0 \to L(s,c) \\ \Gamma^2(T) = \forall\delta.(\delta \geq 0 \wedge s' = s - gc\delta - \frac{1}{2}g\delta^2 \wedge c' = c + \delta \wedge s' \geq 0 \to \\ \qquad s' \geq 0 \wedge s' = 0 \to L(s',c')) \\ \Gamma^3(T) = \forall\delta.(\delta \geq 0 \wedge s' = s - gc\delta - \frac{1}{2}g\delta^2 \wedge c' = c + \delta \wedge s' \geq 0 \to \\ \qquad \forall\delta'.(\delta' \geq 0 \wedge s'' = s' - gc'\delta' - \frac{1}{2}g\delta'^2 \wedge c'' = c' + \delta' \wedge s'' \geq 0 \to \\ \qquad s'' \geq 0 \wedge s'' = 0 \to L(s'',c'')) \end{array} \tag{5}$$

It is easy to see that in Γ^3 the computation terminates (because $\Gamma^3 \to \Gamma^2$). Hence, with the insertion of the initial condition $F(h,0)$, the result is:

$$\forall\delta \ \left(\delta \geq 0 \wedge s' = h - \frac{1}{2}g\delta^2 \wedge c' = \delta \wedge s' \geq 0 \to c' > k \wedge s' = 0 \to L(s',c')\right),$$

where c' and s' are universally quantified variables. This can be simplified to

$$c' \geq 0 \wedge s' = h - \frac{1}{2}gc'^2 \wedge s' \geq 0 \to c' > k \wedge s' = 0 \to L(s',c').$$

Further simplification leads to

$$c' \geq 0 \wedge \sqrt{\frac{2h}{g}} \geq c' \to c' > k \wedge c' = \sqrt{\frac{2h}{g}} \to L(0,c') \tag{6}$$

At this stage it would be necessary to eliminate L as well. In fact this is very simple and therefore is omitted here. From formula 6 we can extract a constraint on the variable k, namely: $k < \sqrt{\frac{2h}{g}}$. And since we had a look at the negation of the property to be proved, we finally end up with the result that the stone is landed for all values of the clock of at least $\sqrt{\frac{2h}{g}}$. Thus the moment of landing is exactly when $c = \sqrt{\frac{2h}{g}}$.

We now point to three relevant mathematical subtasks that occur in the context of the DMC approach:

(1) In the example we have to solve the differential equations $\dot{c}(t) = 1$ and $\dot{s}(t) = -gt$. The solution is employed in the formalisation of the reachability theory.

While this is trivial in our hybrid system, the solution of non-linear differential equations is generally complex and not easily computable.

(2) The DMC approach stepwise eliminates the second order state predicates and thereby generates sets of constraints. As indicated above a subsequent task is then to check the consistency of the generated constraint sets in order to show that a model exists. In our example above, for instance, we are interested in constraints like:

$$\delta \geq 0 \wedge s' = s - gc\delta - \tfrac{1}{2}g\delta^2 \wedge$$
$$c' = c + \delta \wedge s' \geq 0 \rightarrow c' > k$$

The constraint variables are c and s,while δ, c', s' universally quantified parameters coming from the reachability theory.

(3) Generally the fix-point computations involved in the DMC approach are not as trivial as in the example here. The detection of fix-points can then be supported by checking the subsumption of the constraint sets of single iterations in the fix-point computation.

The overall picture is that the verification tool, implementing the deduction-based verification approach, is processing the main steps involved. This, for instance, includes the formalisation of the reachability theory and the stepwise elimination of second order variables. The verification tool is also responsible for the generation and appropriate formulation of the concrete mathematical service requests illustrated in (1) – (3) and for passing them to the MathWeb-SB. A verification example that illustrates the work-sharing aspects of the sketched verification approach in more detail is given in [4]. As long as the verification tool is not fully implemented its tasks or parts of its tasks will be simulated by hand.

In the remainder of this paper we will concentrate on the mathematical service network MathWeb-SB, which is our network infrastructure of choice. We will also present candidate systems for supporting the mathematical subtasks we are interested in.

3 The MathWeb Software Bus

The MathWeb Software Bus (MathWeb-SB) [20] for distributed automated theorem proving supports the connection of a wide range of *mathematical services* by a common software bus. The MathWeb-SB provides the functionality to turn existing theorem proving systems, computer algebra systems, and miscellaneous tools into mathematical services that are homogeneously integrated into a proof development environment.

The MathWeb-SB is implemented in Mozart Oz [11], a multi-paradigm object-oriented programming language which fully supports concurrent and distributed programming and allows to simply distribute applications over the Internet. The services of the MathWeb-SB are used permanently by client applications, such as the ΩMEGA system [10]. The MathWeb-SB currently integrates many different reasoning and computation systems, like, for instance, automated theorem provers (e.g., Otter, Spass, etc.) and computer algebra systems (CASs) (e.g., Maple, and GAP). Fig. 2 shows parts of the MathWeb-SB as it is currently running. In the MathWeb-SB, *service servers* offer the mathematical services (e.g., an ATP, or a CAS) to their local MathWeb-SB broker. MathWeb brokers register and unregister to other brokers, so called *remote brokers*, running in the Internet and therefore build a dynamic web of brokers.

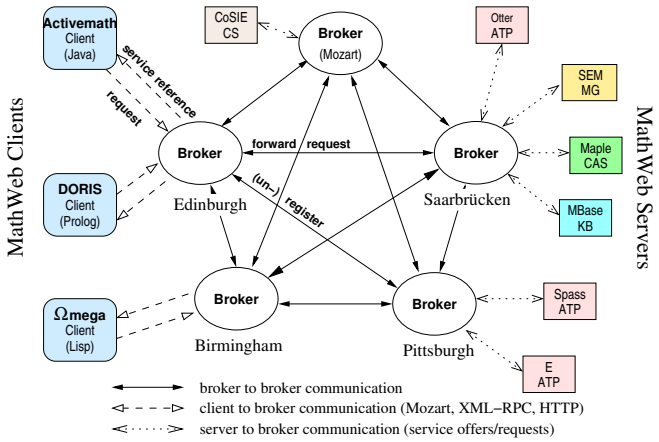


Fig. 2. The MathWeb Software Bus

Client applications, like the ΩMEGA system or a CGI-script, connect to one of the MathWeb-SB brokers and request services. If the requested service is not offered by a local server, the broker forwards the request to all remote brokers. If the requested service is found, the client application receives a reference to a newly created *service object* and can directly send messages to the object. The MathWeb-SB currently offers three interfaces to connect to a broker, namely Mozart's distributed programming interface, CGI-script access via an HTTP server, and access via XML-RPC.

3.1 Solving Differential Equations

Amongst the available Computer Algebra Systems in the MathWeb-SB we choose Maple [6] as a first candidate to support our tasks. Maple is a mathematical problem-solving environment that supports a wide variety of mathematical operations such as numerical analysis, symbolic algebra, and graphics. We intend

to use the module for differential equations, especially the `dsolve` function, that is a part of MAPLE. This module has the capability to solve Ordinary Differential Equations (ODEs). The `dsolve` function is good in solving linear differential equations very efficiently and provides good results, but it is quite weak in attacking non-linear systems. A better candidate for the non-linear case is the module NODES (Non linear Ordinary Differential Equations Solver) [8], which has been developed in context of the European ESPRIT contact group CATHODE. The NODES module is implemented in the MAPLE programming language and is specialised in the analysis of systems of non-linear ODEs. It is based on the quasi-monomial transformation theory [24]. In that theory, a system of ODEs is represented by a couple of matrices and only these are manipulated. NODES identifies values of the parameters in the relative matrix representation corresponding to the integrability property of the ODE system and builds the associated first integrals.

3.2 Checking Consistency of Constraints

The Rewrite and Decision procedure Laboratory (RDL) [1] simplifies clauses in a quantifier-free first-order logic with equality using a tight integration between rewriting and decision procedures. RDL is based on CCR (Constraint Contextual Rewriting) [2], a formally specified integration schema between (ordered) conditional rewriting and a satisfiability decision procedure. As a consequence, RDL is sound, terminating and fully automatic.

RDL is an open system which can be modularly extended with new decision procedures provided these offer certain interface functionalities. In its current version, RDL offers *'plug and play'* decision procedures for the theories of Universal Presburger Arithmetic over Integers (UPAI), Universal Theory of Equality (UTE), and UPAI extended with uninterpreted function symbols. Last but not least, RDL implements instances of a generic extension schema for decision procedures. The key ingredient of such a schema is a lemma speculation mechanism which 'reduces' the satisfiability problem of a given theory to the satisfiability problem of one of its sub-theories for which a decision procedure is available. In the following we explain in few words how the lemma speculation works on constraint sets.

In the context of our subtask, subsumption and checking of constraints can be attacked in two ways. If the set of constraints looks like a set of polynomial or trigonometric functions we can simplify them using highly efficient arithmetic libraries. Due to this, we can handle constraints containing trigonometric functions such as $\sin(x)$, $\cos(x)$, etc. In case that arithmetic is unable to simplify the constraint set, we can attack the problem using the quantifier elimination approach. The mechanism that allows RDL to decide which is the best choice for the solving of the problem is Lemma Speculation. Here we sketch briefly how this mechanism works for constraint subsumption.

Lemma speculation. The goal of this mechanism is to feed the decision procedure with new facts about function symbols which are otherwise uninterpreted

in the theory T decided by the decision procedure. In other words, it inspects the context C and returns a set of ground facts entailed by C using T as the background theory. In RDL there are three kinds of lemma speculation: the simplest is `augment` that finds instances of the conclusions among the conditional lemmas which can promote further inference steps in the decision procedure; an improvement of the `augment` method is `affinize` that implements the 'on the fly' generation of lemmas about multiplication over integers. Affinization is particularly useful for non-linear inequalities and doesn't require any user intervention. As most powerful choice, there is the combination of the two mechanisms mentioned above. RDL combines augmentation and affinization by considering the function symbols occurring in the context C. For example, the top-most function symbol of the largest literal in C triggers the invocation of either mechanisms.

4 Work Plan

We will investigate whether the sketched approach is applicable to industrial-strength examples. To gain evidence for this we want to pursue case studies like air traffic management [21, 22] and the steam-boiler problem [12]. Starting with an appropriate representation of the automata we will apply the DMC approach and identify the concrete instances of the mathematical subtasks described above. We then analyse whether and how these subtasks can actually be attacked by the systems already available in the MathWeb-SB. The aim then is to suitably model the subtasks as service requests to the MathWeb-SB. We might possibly have to integrate new systems into the MathWeb-SB like, for instance, the RDL system.

In the current implementation of the MathWeb-SB, the services requested by client applications are whole reasoning systems (e.g. the CAS MAPLE or ATPs such as OTTER). The service objects offer interface methods for using the system's reasoning capabilities, for instance the method `eval` in the case of CASs or `prove` in the case of ATPs. For our work, we have to extend the MathWeb-SB such that also abstract reasoning services, e.g. solving differential equations, can be defined, offered to MathWeb brokers, and requested by clients. We plan an implementation of abstract reasoning service as new interface methods of the services objects. We also intend to use a service description language to describe reasoning services independent of a concrete implementation. This language will be based on XML-standards WSDL [7], OPENMATH [5], and OMDOC [19]. Abstract service descriptions can then be mapped to interface method calls.

References

1. A. Armando, L. Compagna, and S. Ranise. System description: Rdl–rewrite and decision procedure laboratory. In *International Joint Conference on Automated Reasoning (IJCAR2001)*, 2001.
2. A. Armando and S. Ranise. Constraint contextual rewriting. In R. Caferra and G. Salzer, editors, *Proceedings of the 2nd International Workshop on First Order Theorem Proving, FTP'98, Vienna (Austria)*, pages 65–75, 1998.

3. A. Balluchi, M. Di Benedetto, C. Pinello, C. Rossi, and A. Sangiovanni-Vincentelli. Hybrid control in automotive applications: the cut-off control, 1999.
4. C. Benzmüller, C. Giromini, and A. Nonnengart. Symbolic verification of hybrid systems supported by mathematical services. In *Proceedings of the Calculemus Symposium 2002*, 2002. forthcoming.
5. O. Caprotti and A. M. Cohen. Draft of the Open Math standard. The Open Math Society, `http://www.nag.co.uk/projects/OpenMath/omstd/`, 1998.
6. B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, and S. M. Watt. *First leaves: a tutorial introduction to Maple V*. Springer Verlag, Berlin, 1992.
7. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Service Description Language. W3C Recommendation 1.1, World Wide Web Consortium, 2001. Available at `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`.
8. M. Codutti. Nodes : Non linear ordinary differential equations solver. In *Proceedings of ISSAC'92 (International Symposium on Symbolic and Algebraic Computation)*, 1992.
9. S. Engell, S. Kowalewski, C. Schulz, and O. Stursberg. analysis and optimization of continuous-discrete interactions in chemical processing plants.
10. C. Benzmüller et al. ΩMEGA: Towards a mathematical assistant. In *Proceedings of the 14^{th} International Conference on Automated Deduction (CADE–14)*, pages 252–255. Springer Verlag, Berlin, 1997.
11. The Oz group. The mozart programming system. http://www.mozart-oz.org/.
12. T. Henzinger and H. Wong-Toi. Using hytech to synthesize control parameters for a steam boiler, 1996.
13. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):110–122, 1997.
14. T. A. Henzinger, B. Horowitz, R. Majumdar, and H. Wong-Toi. Beyond HYTECH: Hybrid systems analysis using interval numerical methods. In *HSCC*, pages 130–144, 2000.
15. T.A. Henzinger and P.H. Ho. HyTech: The Cornell Hybrid Technology Tool. In P. Antsaklis, A. Nerode, W. Kohn, and S. Sastry, editors, *Hybrid Systems II*, Lecture Notes in Computer Science 999, pages 265–293. Springer-Verlag, 1995.
16. T.A. Henzinger and P.H. Ho. A note on abstract-interpretation strategies for hybrid automata. In P. Antsaklis, A. Nerode, W. Kohn, and S. Sastry, editors, *Hybrid Systems II*, Lecture Notes in Computer Science 999, pages 252–264. Springer-Verlag, 1995.
17. T.A. Henzinger and H. Wong-Toi. Linear phase-portrait approximations for nonlinear hybrid systems. In R. Alur, T.A. Henzinger, and E.D. Sontag, editors, *Hybrid Systems III*, Lecture Notes in Computer Science 1066, pages 377–388. Springer-Verlag, 1996.
18. R. Horowitz and P. Varaiya. Design of an automated highway system. Proceedings of the IEEE. This issue.
19. M. Kohlhase. OMDoc: Towards an internet standard for the administration, distribution and teaching of mathematical knowledge. In *Proceedings of AI and Symbolic Computation, AISC-2000*, LNAI. Springer Verlag, 2000. Forthcoming.
20. M. Kohlhase and J. Zimmer. System description: The Mathweb Software Bus for Distributed Mathmatical Reasoning; to appear.
21. C. Livadas, J. Lygeros, and N. Lynch. High-level modelling and analysis of tcas, 1999.

22. J. Lygeros, G. J. Pappas, and S. Sastry. An approach to the verification of the center-TRACON automation system. In *HSCC*, pages 289–304, 1998.
23. A. Nonnengart. A deductive model checking approach for hybrid systems. Technical Report MPI-I-1999-2-006, Max-Planck-Institute for Computer Science, Saarbrücken, Germany, November 1999. Available via http://www.mpi-sb.mpg.de/.
24. B.-C.-V. Ung. Combinatorial identities for series of quasi-symmetric functions.

Session:
Logical Approaches

A Trustworthy Proof Checker

Andrew W. Appel*
Princeton University

Neophytos Michael*
Princeton University

Aaron Stump†
Stanford University

Roberto Virga*
Princeton University

June 28, 2002

Abstract

Proof-Carrying Code (PCC) and other applications in computer security require machine-checkable proofs of properties of machine-language programs. The main advantage of the PCC approach is that the amount of code that must be explicitly trusted is very small: it consists of the logic in which predicates and proofs are expressed, the safety predicate, and the proof checker. We have built a minimal-TCB checker, and we explain its design principles, and the representation issues of the logic, safety predicate, and safety proofs. We show that the trusted code in such a system can indeed be very small. In our current system the TCB is less than 2,700 lines of code (an order of magnitude smaller even than other PCC systems) which adds to our confidence of its correctness.

1 Introduction

Machine-verified proofs have applications in computer security, program verification, and the formalization of mathematics. We are particularly interested in security applications such as proof-carrying code, in which an untrusted program provider must send a proof that the machine-language program is safe to execute; or proof-carrying authorization, in which an untrusted client must send a proof that some desired server operation conforms to an authorization policy. The recipient doesn't trust the proof provider, so the recipient's proof checker is an essential component of the trusted computing base: a bug in the proof checker can be a security hole in the larger system. Therefore, the checker must be trustworthy: it must be small, simple, readable, and based on well-understood engineering and mathematical principles.

In contrast, theorem provers are often large and ugly, as required by the incompleteness results of Gödel and Turing: no prover of bounded size is sufficiently general, but one can always hack more features into the prover until it proves the desired class of theorems. It is difficult to fully trust such software, so some proving systems use technical means to ensure that buggy provers cannot produce invalid proofs: the abstract data type *theorem* of LCF [14], or the proof-witness objects of Coq [8] or Twelf [20]. With these means, only a small part of a large system must be examined and trusted.

How large is the proof checker that must be examined and trusted? To answer this question we have tried the experiment of constructing and measuring the *smallest possible* useful proof checker for some real application. Our checker receives, checks the safety of, and executes, proof-carrying code: machine code for the Sparc with an accompanying proof of safety. The proof is in higher-order logic represented in LF notation.

This checker would also be directly useful for proof-carrying authorization [3, 9], that is, checking proofs of authentication and permission according to some distributed policy.

A useful measure of the effort required to examine, understand, and trust a program is its size in (non-blank, non-comment) lines of source code. Although there may be much variation in effort and complexity per line of code, a crude quantitative measure is better than none. It is also necessary to count, or otherwise account for, any compiler, libraries, or supporting software used to execute the program. We address this issue explicitly by avoiding the use of libraries and by making the checker small enough so that it can be examined in machine language.

The *trusted computing base* (TCB) of a proof-carrying code system consists of all code that must be explicitly trusted as correct by the user of the system. In our case the TCB consists of two pieces: first, the specification of the safety predicate in higher-order logic, and second, the proof checker, a small C program that checks proofs, loads, and executes safe programs.

In his investigation of Java-enabled browsers [11], Ed Felten found that the first-generation implementations averaged one security-relevant bug per 3,000 lines of source code [13]. These browsers, as mobile-code host platforms

*This research was supported in part by DARPA award F30602-99-1-0519.

†This research was supported DARPA/Air Force contract F33615-00-C-1693 and NSF contract CCR-9806889.

that depend on static checking for security, exemplify the kind of application for which proof-carrying code is well suited. Wang and Appel [7] measured the TCBs of various Java Virtual Machines at between 50,000 and 200,000 lines of code. The SpecialJ JVM [10] uses proof-carrying code to reduce the TCB to 36,000 lines.

In this work, we show how to reduce the size of the TCB to under 2,700 lines, and by basing those lines on a well understood logical framework, we have produced a checker which is small enough so that it can be manually verified; and as such it can be relied upon to accept only valid proofs. Since this small checker "knows" only about machine instructions, and nothing about the programming language being compiled and its type system, the semantic techniques for generating the proofs that the TCB will check can be involved and complex [2], but the checker doesn't.

2 The LF logical framework

For a proof checker to be simple and correct, it is helpful to use a well designed and well understood representation for logics, theorems, and proofs. We use the LF logical framework.

LF [15] provides a means for defining and presenting logics. The framework is general enough to represent a great number of logics of interest in mathematics and computer science (for instance: first-order, higher-order, intuitionistic, classical, modal, temporal, relevant and linear logics, and others). The framework is based on a general treatment of syntax, rules, and proofs by means of a typed first-order λ-calculus with dependent types. The LF type system has three levels of terms: objects, types, and kinds. Types classify objects and kinds classify families of types. The formal notion of definitional equality is taken to be $\beta\eta$-conversion.

A logical system is presented by a signature, which assigns types and kinds to a finite set of constants that represent its syntax, its judgments, and its rule schemes. The LF type system ensures that object-logic terms are well formed. At the proof level, the system is based on the *judgments-as-types* principle: judgments are represented as types, and proofs are represented as terms whose type is the representation of the theorem they prove. Thus, there is a correspondence between type-checked terms and theorems of the object logic. In this way proof checking of the object logic is reduced to type checking of the LF terms.

For developing our proofs, we use Twelf [20], an implementation of LF by Frank Pfenning and his students. Twelf is a sophisticated system with many useful features: in addition to an LF type checker, it contains a type-reconstruction algorithm that permits users to omit many explicit parameters, a proof-search algorithm, constraint regimes (e.g., linear programming over the exact rational numbers), mode analysis of parameters, a meta-theorem prover, a pretty-printer, a module system, a configuration system, an interactive Emacs mode, and more. We have found many of these features useful in proof development, but Twelf is certainly not a minimal proof checker. However, since Twelf does construct explicit proof objects internally, we can extract these objects to send to our minimal checker.

In LF one declares the operators, axioms, and inference rules of an *object logic* as constructors. For example, we can declare a fragment of first-order logic with the type `form` for formulas and a dependent type constructor `pf` for proofs, so that for any formula `A`, the type `pf(A)` contains values that are proofs of `A`. Then, we can declare an "implies" constructor `imp` (infix, so it appears between its arguments), so that if `A` and `B` are formulas then so is `A imp B`. Finally, we can define introduction and elimination rules for `imp`.

```
form : type.
pf   : form -> type.
imp  : form -> form -> form. %in-
fix right 10 imp.
imp_i: (pf A -> pf B) -> pf (A imp B).
imp_e: pf (A imp B) -> pf A -> pf B.
```

All the above are defined as constructors. In general, constructors have the form $name : \tau$ and declare that $name$ is a value of type τ.

It is easy to declare inconsistent object-logic constructors. For example, `invalid: pf A` is a constructor that acts as a proof of any formula, so using it we could easily prove the false proposition:

```
logic_inconsistent : pf (false) = invalid.
```

So the object logic should be designed carefully and must be trusted.

Once the object logic is defined, theorems can be proved. We can prove for instance that implication is transitive:

```
imp_trans:
  pf (A imp B) -> pf (B imp C) -
> pf (A imp C) =
    [p1: pf (A imp B)][p2: pf (B imp C)]
     imp_i [p3: pf A] imp_e p2 (imp_e p1 p3).
```

In general, definitions (including predicates and theorems) have the form $name : \tau = exp$, which means that $name$ is now to stand for the value exp whose type is τ. In this example, the exp is a function with formal parameters `p1` and `p2`, and with body `imp_i [p3] imp_e p2 (imp_e p1 p3)`.

Definitions need not be trusted, because the type-checker can verify whether exp does have type τ. In general, if a proof checker is to check the proof P of theorem T in a logic $\mathcal{L}$, then the constructors (operators and axioms) of $\mathcal{L}$ must be given to the checker in a trusted way (i.e., the adversary must not be free to install inconsistent axioms). The statement of T must also be trusted

(i.e., the adversary must not be free to substitute an irrelevant or vacuous theorem). The adversary provides only the proof P, and then the checker does the proof checking (i.e., it type-checks in the LF type system the definition $t : T = P$, for some arbitrary name t).

3 Application: Proof-carrying code

Our checker is intended to serve a purpose: to check safety theorems about machine-language programs. It is important to include application-specific portions of the checker in our measurements to ensure that we have adequately addressed all issues relating to interfacing to the real world.

The most important real-world-interface issue is, "is the proved theorem meaningful?" An accurate checker does no good if it checks the wrong theorem. As we will explain, the specification of the safety theorem is larger than all the other components of our checker combined!

Given a machine-language program P, that is, a sequence of integers that code for machine instructions (on the Sparc, in our case), the theorem is, "when run on the Sparc, P never executes an illegal instruction, nor reads or writes from memory outside a given range of addresses." To formalize this theorem it is necessary to formalize a description of instruction execution on the Sparc processor. We do this in higher-order logic augmented with arithmetic.

In our model [16], a machine state (r, m) comprises a *register bank* (r), and a *memory* (m), each of which is a function from integers (register numbers and addresses) to integers (contents). Every register of the instruction-set architecture (ISA) must be assigned a number in the register bank: the general registers, the floating-point registers, the condition codes, and the program counter. Where the ISA does not specify a number (such as for the PC) or when the numbers for two registers conflict (such as for the floating point and integer registers) we use an arbitrary unused index.

A single step of the machine is the execution of one instruction. We can specify instruction execution by giving a step relation $(r, m) \mapsto (r', m')$ that maps the prior state (r, m) to the next state (r', m') that holds after the execution of the machine instruction.

For example, to describe the "add" instruction $r_1 \leftarrow r_2 + r_3$ we might start by writing,

$$(r, m) \mapsto (r', m') \;\equiv\; r'(1) = r(2) + r(3) \\ \wedge\, (\forall x \neq 1.\, r'(x) = r(x)) \;\wedge\; m' = m$$

In fact, we can parameterize the above on the three registers involved and define $\mathrm{add}(i, j, k)$ as the following predicate on four arguments (r, m, r', m'):

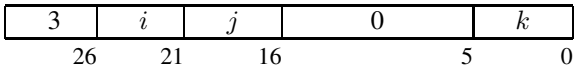
$$\mathrm{add}(i, j, k) \;\equiv \\ \lambda r, m, r', m'.\; r'(i) = r(j) + r(k) \\ \wedge\, (\forall x \neq i.\, r'(x) = r(x)) \;\wedge\; m' = m$$

Similarly, for the "load" instruction $r_i \leftarrow m[r_j + c]$ we define its semantics to be the predicate:

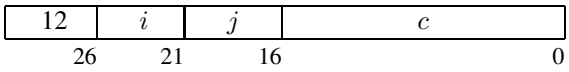
$$\mathrm{load}(i, j, c) \;\equiv \\ \lambda r, m, r', m'.\; r'(i) = m(r(j) + c) \\ \wedge\, (\forall x \neq i.\, r'(x) = r(x)) \;\wedge\; m' = m$$

To enforce memory safety policies, we will modify the definition of $\mathrm{load}(i, j, c)$ to require that the loaded address is legal [2], but we omit those details here.

But we must also take into account instruction fetch and decode. Suppose, for example, that the "add" instruction is encoded as a 32-bit word, containing a 6-bit field with opcode 3 denoting *add,* a 5-bit field denoting the destination register i, and 5-bit fields denoting the source registers j, k:



The "load" instruction might be encoded as:



Then we can say that some number w decodes to an instruction *instr* iff,

$$\mathrm{decode}(w, \mathit{instr}) \;\equiv \\ (\exists i, j, k. \\ \quad 0 \leq i < 2^5 \;\wedge\; 0 \leq j < 2^5 \;\wedge\; 0 \leq k < 2^5 \;\wedge \\ \quad w = 3 \cdot 2^{26} + i \cdot 2^{21} + j \cdot 2^{16} + k \cdot 2^0 \;\wedge \\ \quad \mathit{instr} = \mathrm{add}(i, j, k)) \\ \vee\, (\exists i, j, c. \\ \quad 0 \leq i < 2^5 \;\wedge\; 0 \leq j < 2^5 \;\wedge\; 0 \leq c < 2^{16} \;\wedge \\ \quad w = 12 \cdot 2^{26} + i \cdot 2^{21} + j \cdot 2^{16} + c \cdot 2^0 \;\wedge \\ \quad \mathit{instr} = \mathrm{load}(i, j, c)) \\ \vee\; \ldots$$

with the ellipsis denoting the many other instructions of the machine, which must also be specified in this formula.

We have shown [16] how to scale this idea up to the instruction set of a real machine. Real machines have large but semi-regular instruction sets; instead of a single global disjunction, the decode relation can be factored into operands, addressing modes, and so on. Real machines don't use integer arithmetic, they use modular arithmetic, which can itself be specified in our higher-order logic. Some real machines have multiple program counters (e.g., Sparc) or variable-length instructions (e.g., Pentium), and these can also be accommodated.

Our description of the decode relation is heavily factored by higher-order predicates (this would not be possible without higher-order logic). We have specified the execution behavior of a large subset of the Sparc architecture, and we have built a prototype proof-generating compiler that targets that subset. For proof-carrying code, it is sufficient to specify a subset of the machine architecture; any unspecified instruction will be treated by the safety policy as illegal. While this may be inconvenient for compilers

that want to generate that instruction, it does ensure that safety cannot be compromised.

4 Specifying safety

Our step relation $(r, m) \mapsto (r', m')$ is deliberately partial; some states have no successor state. In these states the program counter $r(\text{PC})$ points to an illegal instruction. Using this partial step relation, we can define safety. A safe program is one that will never execute an illegal instruction; that is, a given state is safe if, for any state reachable in the Kleene closure of the step relation, there is a successor state:

$$\text{safe-state}(r, m) \equiv \forall r', m'.\ (r, m) \mapsto^* (r', m') \Rightarrow \exists r'', m''.\ (r', m') \mapsto (r'', m'')$$

A program is just a sequence of integers (each representing a machine instruction); we say that a program p is loaded at a location l in memory m if

$$\text{loaded}(p, m, l) \equiv \forall i \in \text{dom}(p).\ m(i + l) = p(i)$$

Finally (assuming that programs are written in position-independent code), a program is *safe* if, no matter where we load it in memory, we get a safe state:

$$\text{safe}(p) \equiv \forall r, m, l.\ \text{loaded}(p, m, l) \wedge r(\text{PC}) = l \Rightarrow \text{safe-state}(r, m)$$

Let `;` be a "cons" operator for sequences of integers (easily definable in HOL); then for some program `8420; 2837; 2938; 2384; nil` the safety theorem is simply:

```
safe (8420; 2837; 2938; 2384; nil)
```

and, given a proof P, the LF definition that must be type-checked is:

```
t: pf(safe(8420; 2837; 2938; 2384; nil)) = P.
```

Though we wrote in section 2 that definitions need not be trusted because they can be type-checked, this is not strictly true. Any definition used in the statement of the theorem must be trusted, because the wrong definition will lead to the proof of the wrong theorem. Thus, all the definitions leading up to the definition of `safe` (including `add`, `load`, `safe-state`, `step`, etc.) must be part of the trusted checker. Since we have approximately 1,600 lines of such definitions, and they are a component of our "minimal" checker, one of the most important issues we faced is the representation of these definitions; we discuss this in Section 7.

On the other hand, a large proof will contain hundreds of internal definitions. These are predicates and internal lemmas of the proof (not of the statement of the theorem), and are at the discretion of the proof provider. Since each is type checked by the checker before it is used in further definitions and proofs, they don't need to be trusted.

In the table below we show the various pieces needed for the specification of the safety theorem in our logic. Every piece in this table is part of the TCB. The first two lines show the size of the logical and arithmetic connectives (in which theorems are specified) as well as the size of the logical and arithmetic axioms (using which theorems are proved). The Sparc specification has two components, a "syntactic" part (the decode relation) and a semantic part (the definitions of *add, load,* etc.); these are shown in the next two lines. The size of the safety predicate is shown last.

Safety Specification	*Lines*	*Definitions*
Logic	135	61
Arithmetic	160	94
Machine Syntax	460	334
Machine Semantics	1,005	692
Safety Predicate	105	25
Total	1,865	1,206

¿From this point on we will refer to everything in the table as the *safety specification* or simply the *specification*.

5 Eliminating redundancy

Typically an LF signature will contain much redundant information. Consider for example the rule for `imp` introduction presented previously; in fully explicit form, their representation in LF is as follows:

```
imp_i : {A : form}{B : form}
        (pf A -> pf B) -> pf (A imp B).
```

The fact that both `A` and `B` are formulas can be easily inferred by the fact they are given as arguments to the constructor `imp`, which has been previously declared as an operator over formulas.

On the one hand, eliminating redundancy from the representation of proofs benefits both proof size and type-checking time. On the other hand, it requires performing term reconstruction, and thus it may dramatically increase the complexity of type checking, driving us away from our goal of building a minimal checker.

Twelf deals with redundancy by allowing the user to declare some parameters as *implicit*. More precisely, all variables which are not quantified in the declaration are automatically assumed implicit. Whenever an operator is used, Twelf's term reconstruction will try to determine the correct substitution for all its implicit arguments. For example, in type-checking the lemma

```
imp_refl: pf (A imp A) = imp_i ([p : pf A] p).
```

Twelf will automatically reconstruct the two implicit arguments of `imp_i` to be both equal to `A`.

While Twelf's notion of implicit arguments is effective in eliminating most of the redundancy, type reconstruction adds considerable complexity to the system. Another

drawback of Twelf's type reconstruction is its reliance on higher-order unification, which is undecidable. Because of this, type checking of some valid proofs may fail.

Since full type reconstruction is too complex to use in a trusted checker, one might think of sending fully explicit LF proof terms; but a fully explicit proof in Twelf syntax can be exponentially larger than its implicit representation. To avoid these problems, Necula's LF_i [18] uses partial type reconstruction and a simple algorithm to determine which of the arguments can be made implicit. Implicit arguments are omitted in the representation, and replaced by placeholders. Oracle-based checking [19] reduces the proof size even further by allowing the erasure of subterms whose reconstruction is not uniquely determined. Specifically, in cases when the reconstruction of a subterm is not unique, but there is a finite (and usually small) list of candidates, it stores an oracle for the right candidate number instead of storing the entire subterm.

These techniques use clever syntactic representations of proofs that minimize proof size; checking these representations is not as complex as full Twelf-style type reconstruction, but is still more complex than is appropriate for a minimal proof checker. We are willing to tolerate somewhat larger proofs in exchange for a really simple checking algorithm. Instead of using a syntactic representation of proofs, we avoid the need for the checker to parse proofs by using a data-structure reprentation. However, we still need to avoid exponential blowups, so we reduce redundancy by structure sharing. Therefore, we represent and transmit proofs as LF terms in the form of directed acyclic graphs (DAGs), with structure sharing of common subexpressions to avoid exponential blowup.

A node in the DAG can be one of ten possible types: one for kinds, five for ordinary LF terms, and four for arithmetic expressions. Each node may store up to three integers, `arg1`, `arg2`, and `type`. This last one, if present, will always point to the sub-DAG representing the type of the expression.

	arg1	arg2	type	
n	U	U	U	kind
c	U	U	M	constant
v	M	M	M	variable
a	M	M	O	application
p	M	M	O	product
l	M	M	O	abstraction
#	M	U	O	number
+	M	M	O	addition proof object
*	M	M	O	mult proof object
/	M	M	O	div proof object

M = mandatory, O = optional, U = unused

The content of `arg1` and `arg2` is used in different ways for different node types. For all nodes representing arithmetic expressions ('#', '+', '*', and '/'), they contain integer values. For products and abstractions ('p' and 'l'), `arg1` points to the bound variable, and `arg2` to the term where the binding takes place. For variable nodes ('v'), they are used to make sure that the variable always occurs within the scope of a quantifier. For application nodes ('a'), they point to the function and its argument, respectively. Finally, constant declaration nodes ('c'), and kind declaration nodes ('n') use neither.

For a concrete example, consider the LF signature:

```
form : type.
pf   : form -> type.
imp  : form -> form -> form.
```

We present below the DAG representation of this signature. We "flattened" the DAG into a numbered list, and, for clarity, we also added a comment on the right showing the corresponding LF term.

```
 1| n 0 0 0   ; type Kind
 2| c 0 0 1   ; form: type
 3| v 0 0 2   ; x: form
 4| p 3 1 0   ; {x: form} type
 5| c 0 0 4   ; pf: {x: form} type
 6| v 0 0 2   ; y: form
 7| p 6 2 0   ; {y: form} form
 8| v 0 0 2   ; x: form
 9| p 8 7 0   ; {x: form}{y: form} form
10| c 0 0 9   ; imp: {x: form}{y: form} form
```

6 Dealing with arithmetic

Since our proofs reason about encodings of machine instructions (opcode calculations) and integer values manipulated by programs, the problem of representing arithmetic within our system is a critical one. A purely logical representation based on 0, successor and predecessor is not suitable to us, since it would cause proof size to explode.

The latest releases of Twelf offer extensions that deal natively with infinite-precision integers and rationals. While these extensions are very powerful and convenient to use, they offer far more than we need, and because of their generality they have a very complex implementation (the rational extension alone is 1,950 lines of Standard ML). What we would like for our checker is an extension built in the same spirit as those, but much simpler and lighter.

We require two properties from such an extension:

1. LF terms for all the numbers we use; moreover, the size of the LF term for n should be constant and independent of n.

2. Proof objects for single-operation arithmetic facts such as "$10 + 2 = 12$"; again, we require that such proof objects have constant size.

Our arithmetic extensions to the checker are the smallest and simplest ones to satisfy (1) and (2) above. We add the `word32` type to the TCB, (representing integers in the range $[0, 2^{32} - 1]$) as well as the following axioms:

```
+ : word32 -> word32 -> word32 -> type.
* : word32 -> word32 -> word32 -> type.
/ : word32 -> word32 -> word32 -> type.
```

We also modify the checker to accept arithmetic terms such as:

```
456+25 : + 456 25 481.
32*4   : * 32 4 128.
```

This extension does not modify in any way the standard LF type checking: we could have obtained the same result (although much more inefficiently) if we added all these constants to the trusted LF signature by hand. However, granting them special treatment allowed us to save literally millions of lines in the axioms in exchange for an extra 55 lines in the checker.

To embed and use these new constants in our object logic, we also declare:

```
c: word32 -> tm num.

eval_plus: + A B C ->
   pf (eq (plus (c A) (c B)) (c C)).

eval_times: * A B C ->
   pf (eq (times (c A) (c B)) (c C)).

eval_div: / M N Q ->
   pf ((geq (c M) (times (c N) (c Q))) and
       (not (geq (c M) (times (c N)
                         (plus one (c Q)))))).
```

This embedding from `word32` to numbers in our object logic is not surjective. Numbers in our object logic are still unbounded; `word32` merely provides us with handy names for the ones used most often.

With this "glue" to connect object logic to meta logic, numbers and proofs of elementary arithmetic properties, are just terms of size two.

7 Representing axioms and trusted definitions

Since we can represent axioms, theorems, and proofs as DAGs, it might seem that we need neither a parser nor a pretty-printer in our minimal checker. In principle, we could provide our checker with an initial trusted DAG representing the axioms and the theorem to be proved, and then it could receive and check an untrusted DAG representing the proof. The trusted DAG could be represented in the C language as an initialized array of graph nodes.

This might work if we had a very small number of axioms and trusted definitions, and if the statement of the theorem to be proved were very small. We would have to read and trust the initialized-array statements in C, and understand their correspondence to the axioms (etc.) as we would write them in LF notation. For a sufficiently small DAG, this might be simpler than reading and trusting a parser for LF notation.

However, even a small set of operators and axioms (especially once the axioms of arithmetic are included) requires hundreds of graph nodes. In addition, as explained in section 4, our trusted definitions include the machine-instruction step relation of the Sparc processor. These 1,865 lines of Twelf expand to 22,270 DAG nodes. Clearly it is impossible for a human to directly read and trust a graph that large.

Therefore, we require a parser or pretty-printer in the minimal checker; we choose to use a parser. Our C program will parse the 1,865 lines of axioms and trusted definitions, translating the LF notation into DAG nodes. The axioms and definitions are also part of the C program: they are a constant string to which the parser is applied on startup.

This parser is 428 lines of C code; adding these lines to the minimal checker means our minimal checker can use 1,865 lines of LF instead of 22,270 lines of graph-node initializers, clearly a good tradeoff. Our parser accepts valid LF expressions, written in the same syntax used by Twelf. For more details see the full version of the paper [6].

7.1 Encoding higher-order logic in LF

Our encoding of higher-order logic in LF follows that of Harper et al. [15] and is shown in figure 1. The *constructors* generate the syntax of the object logic and the *axioms* generate its proofs. A meta-logical type is `type` and an object-logic type is `tp`. Object-logic types are constructed from `form` (the type of formulas), `num` (the type of integers), and the `arrow` constructor. The LF term `tm` maps an object type to a meta type, so an object-level term of type `T` has type `(tm T)` in the meta logic.

Abstraction in the object logic is expressed by the `lam` term. The term `(lam [x] (F x))` is the object-logic function that maps `x` to `(F x)`. Application for such lambda terms is expressed via the `@` operator. The quantifier `forall` is defined to take as input a meta-level (LF) function of type `(tm T -> tm form)` and produce a `tm form`. The use of the LF functions here makes it easy to perform substitution when a proof of `forall` needs to be discharged, since equality in LF is just $\beta\eta$-conversion.

Notice that most of the standard logical connectives are absent from figure 1. This is because we can produce them as definitions from the constructors we already have. For instance, conjunction can be defined as follows:

```
and = [A][B] forall [C] (A imp B imp C) imp C.
```

It is easy to see that the above formula is equivalent to the standard definition of `and`. We can likewise define introduction and elimination rules for all such logic constructors. These rules are proven as lemmas and need not be trusted. Object-level equality[1] is also easy to define:

[1] The equality predicate `eq` is polymorphic in `T`. Objects `A` and `B` have object type `T` and so they could be `nums`, `forms` or even object level functions (`arrow`

Logic Constructors

```
tp        :  type.
tm        :  tp -> type.
form      :  tp.
arrow     :  tp -> tp -> tp.
pf        :  tm form -> type.
lam       :  (tm T1 -> tm T2) -> tm (T1 arrow T2).
@         :  tm (T1 arrow T2) -> tm T1 -> tm T2.
forall    :  (tm T -> tm form) -> tm form.
imp       :  tm form -> tm form -> tm form.
```

Logic Axioms

```
beta_e    :  pf (P ((lam F) @ X)) -> pf (P (F X)).
beta_i    :  pf (P (F X)) -> pf (P (lam F) @ X).
imp_i     :  (pf A -> pf B) -> pf (A imp B).
imp_e     :  pf (A imp B) -> pf A -> pf B.
forall_i  :  ({X : tm T} pf (A X)) -> pf (forall A).
forall_e  :  pf (forall A) -> {X : tm T} pf (A X).
```

Figure 1: Higher-Order Logic in Twelf

```
eq : tm T -> tm T -> tm form =
 [A][B] forall [P] P @ B imp P @ A.
```

This states that objects `A` and `B` are considered equal iff any predicate `P` that holds on `B` also holds on `A`.

Terms of type `pf A` are terms representing proofs of object formula `A`. Such terms are constructed using the axioms of figure 1. Axioms `beta_i` and `beta_e` are used to prove β-equivalence in the object logic, `imp_i` and `imp_e` transform a meta-level proof function to the object level and vice-versa, and finally, `forall_i` and `forall_e` introduce and eliminate the universal quantifier.

7.2 "Polymorphic" programming in Twelf

ML-style implicit polymorphism allows one to write a function usable at many different argument types, and ML-style type inference does this with a low syntactic overhead. We are writing proofs, not programs, but we would still like to have polymorphic predicates and polymorphic lemmas. LF is not polymorphic in the ML sense, but Harper et al. [15] show how to use LF's dependent type system to get the effect and (most of) the convenience of implicit parametric polymorphism with an encoding trick, which we will illustrate with an example.

Suppose we wish to write the lemma `congr` that would allow us to substitute equals for equals:

```
congr : {H : type -> tm form}
 pf (eq X Z) -> pf (H Z) -> pf (H X) = ...
```

types). The object type `T` is implicit in the sense that when we use the `eq` predicate we do not have to specify it; Twelf can automatically infer it. So internally, the meta-level type of `eq` is not what we have specified above but the following:

```
{T : tp} tm T -> tm T -> tm form.
```

We will have more to say about this in section 7.2.

The lemma states that for any predicate `H`, if `H` holds on `Z` and `Z` = `X` then `H` also holds on `X`. Unfortunately this is ill-typed in LF since LF does not allow polymorphism. Fortunately though, there is way to get polymorphism at the object level. We rewrite `congr` as:

```
congr : {H : tm T -> tm form}
 pf (eq X Z) -> pf (H Z) -> pf (H X) = ...
```

and this is now acceptable to Twelf. Function `H` now judges objects of meta-type `(tm T)` for any object-level type `T`, and so `congr` is now "polymorphic" in `T`. We can apply it on any object-level type, such as `num`, `form`, `num arrow form`, etc. This solution is general enough to allow us to express any polymorphic term or lemma with ease. Axioms `forall_i` and `forall_e` in figure 1 are likewise polymorphic in `T`.

7.3 How to write explicit Twelf

In the definition of lemma `congr` above, we have left out many explicit parameters since Twelf's type-reconstruction algorithm can infer them. The explicit version of the LF term `congr` is:

```
congr : {T : tp}{X : tm T}{Z : tm T}
        {H : tm T -> tm form}
 pf (_eq T X Z) -> pf (H Z) -> pf (H X) = ...
```

(here we also make use of the explicit version of the equality predicate `_eq`). Type reconstruction in Twelf is extremely useful, especially in a large system like ours, where literally hundreds of definitions and lemmas have to be stated and proved.

Object Logic Abstraction/Application

```
fld2 = [T1:tp][T2:tp][T3:tp][T4:tp]
 lam6 (arrow T1 (arrow T2 form))
      (arrow T3 (arrow T2 form))
      (arrow T2 form)
      (arrow T1 (arrow T3 T4))
      T4 T2 form
  [f0][f1][p_pi][icons][ins][w]
  (@ T2 form p_pi w) and
  (exists2 T1 T3 [g0:tm T1] [g1:tm T3]
    (@ T2 form
     (&& T2 (@ T1 (arrow T2 form) f0 g0)
        (@ T3 (arrow T2 form) f1 g1)) w) and
    (eq T4 ins (@ T3 T4 (@ T1 (arrow T3 T4)
                            icons g0) g1))).
```

Meta Logic Abstraction/Application

```
fld2 = [T1:tp][T2:tp][T3:tp][T4:tp]
 [f0][f1][p_pi][icons][ins][w]
 (p_pi w) and
 (exists2 [g0:tm T1][g1:tm T3]
  (f0 g0 && f1 g1) w) and
  (eq ins (icons g0 g1)).
```

Figure 2: Abstraction & Application in the Object versus Meta Logic.

Our safety specification was originally written to take advantage of Twelf's ability to infer missing arguments. Before proof checking can begin, this specification needs to be fed to our proof checker. In choosing then what would be in our TCB we had to decide between the following alternatives:

1. Keep the implicitly-typed specification in the TCB and run it through Twelf to produce an explicit version (with no missing arguments or types). This explicit version would be fed to our proof checker. This approach allows the specification to remain in the implicit style. Also our proof checker would remain simple (with no type reconstruction/inference capabilities) but unfortunately we now have to add to the TCB Twelf's type-reconstruction and unification algorithms, which are about 5,000 lines of ML code.

2. Run the implicitly typed specification through Twelf to get an explicit version. Now instead of trusting the implicit specification and Twelf's type-reconstruction algorithms, we keep them out of the TCB and proceed to manually verify the explicit version. This approach also keeps the checker simple. Unfortunately the explicit specification produced by Twelf explodes in size from 1,700 to 11,000 lines, and thus the code that needs to be verified is huge. The TCB would grow by a lot.

3. Rewrite the trusted definitions in an explicit style. Now we do not need type reconstruction in the TCB (the problem of choice 1), and if the rewrite from the implicit to the explicit style can avoid the size explosion (the problem of choice 2), then we have achieved the best of both worlds.

Only choice 3 was consistent with our goal of a small TCB so we rewrote the trusted definitions in an explicit style while managing to avoid the size explosion. The new safety specification is only 1,865 lines of explicitly-typed Twelf. It contains no terms with implicit arguments and so we do not need a type-reconstruction/type-inference algorithm in the proof checker. The rewrite solves the problem while maintaining the succinctness and brevity of the original TCB, the penalty of the explicit style being an increase in size of 124 lines. The remainder of this section explains the problem in detail and the method we used to bypass it.

To see why there is such an enormous difference in size (1,700 lines vs 11,000) between the implicit specification and its explicit representation generated by Twelf's type-reconstruction algorithm, consider the following example. Let `F` be a two-argument object-level predicate `F : tm (num arrow num arrow form)` (typical case when describing unary operators in an instruction set). When such a predicate is applied, as in `(F @ X @ Y)`, Twelf has to infer the implicit arguments to the two instances of operator `@`. The explicit representation of the application then becomes:

```
@ num form (@ num (num arrow form) F X) Y
```

It is easy to see how the explicit representation explodes in size for terms of higher order. Since the use of higher-order terms was essential in achieving maximal factoring in the machine descriptions [16], the size of the explicit representation quickly becomes unmanageable.

Here is another more concrete example from the `decode` relation of section 3. This one shows how the abstraction operator `lam` suffers from the same problem. The predicate below (given in implicit form) is used in specifying the syntax of all Sparc instructions of two arguments.

```
fld2 = lam6 [f0][f1][p_pi][icons][ins][w]
 p_pi @ w and
 exists2 [g0][g1] (f0 @ g0 && f1 @ g1) @ w and
                  eq ins (icons @ g0 @ g1).
```

Predicates `f0` and `f1` specify the input and the output registers, `p_pi` decides the instruction opcode, `icons` is the

instruction constructor, `ins` is the instruction we are decoding, and `w` is the machine-code word. In explicit form this turns into what we see on the left-hand side of figure 2 – an explicitly typed definition 16 lines long.

The way around this problem is the following: We avoid using object-logic predicates whenever possible. This way we need not specify the types on which object-logic application and abstraction are used. For example, the `fld2` predicate above now becomes what we see on the right-hand side of figure 2. This new predicate has shrunk in size by more than half.

Sometimes moving predicates to the meta-logic is not possible. For instance, we represent *instructions* as predicates from machine states to machine states (see section 3). Such predicates must be in the object logic since we need to be able to use them in quantifiers (`exists [ins : tm instr] ...`). Thus, we face the problem of having to supply all the implicit types when defining and applying such predicates. But since these types are always fixed we can factor the partial applications and avoid the repetition. For example, when defining some Sparc machine instruction as in:

```
i_anyInstr = lam2 [rs : tnum][rd : tnum]
 lam4 registers memory registers memory form
  [r : tregs][m : tmem][r' : tregs][m' : tmem]
   ...
```

we define the predicate `instr_lam` as:

```
instr_lam = lam4 registers memory
                 registers memory form.
```

and then use it in defining each of the 250 or so Sparc instructions as below:

```
i_anyInstr = [rs : tnum][rd : tnum]
 instr_lam [r : tregs][m : tmem]
           [r' : tregs][m' : tmem] ...
```

This technique turns out to be very effective because our machine syntax and semantics specifications were highly factored to begin with [16].

So by moving to the meta logic and by clever factoring we have moved the TCB from implicit to explicit style with only a minor increase in size. Now we don't have to trust a complicated type-reconstruction/type-inference algorithm. What we feed to our proof checker is precisely the set of axioms we explicitly trust.

7.4 The implicit layer

When we are building proofs, we still wish to use the implicit style because of its brevity and convenience. For this reason we have built an implicit layer on top of our explicit TCB. This allows us to write proofs and definitions in the implicit style and LF's $\beta\eta$-conversion takes care of establishing meta-level term equality. For instance, consider the object-logic application operator `_@` given below:

```
_@: {T1 : tp}{T2 : tp} tm (T1 arrow T2) ->
                          tm T1 -> tm T2.
```

In the implicit layer we now define a corresponding application operator `@` in terms of `_@` as follows:

```
@: tm (T1 arrow T2) -> tm T1 -> tm T2 =
    _@ T1 T2.
```

In this term the type variables `T1` and `T2` are implicit and need not be specified when `@` is used. Because `@` is a definition used only in proofs (not in the statement of the safety predicate), it does not have to trusted.

8 The proof checker

The total number of lines of code that form our checker is 2,668. Of these, 1,865 are used to represent the LF signature containing the core axioms and definition, which is stored as a static constant string. The remaining 803 lines of C source code, can be broken down as follows:

Component	*Lines*
Error messaging	14
Input/Output	29
Parser	428
DAG creation and manipulation	111
Type checking and term equality	167
Main program	54
Total	803

We make no use of libraries in any of the components above. Libraries often have bugs, and by avoiding their use we eliminate the possibility that some adversary may exploit one of these bugs to disable or subvert proof checking. However, we do make use of two POSIX calls: `read`, to read the program and proof, and `_exit`, to quit if the proof is invalid. This seems to be the minimum possible use of external libraries.

8.1 Trusting the C compiler

We hasten to point out that these 803 lines of C need to be compiled by a C compiler, and so it would appear that this compiler would need to be included in our TCB. The C compiler could have bugs that may potentially be exploited by an adversary to circumvent proof checking. More dangerously perhaps, the C compiler could have been written by the adversary so while compiling our checker, it could insert a Thompson-style [22] Trojan horse into the executable of the checker.

All proof verification systems suffer from this problem. One solution (as suggested by Pollack [21]) is that of independent checking: write the proof checker in a widely used programming language and then use different compilers of that language to compile the checker. Then, run all your checkers on the proof in question. This is similar

to the way mathematical proofs are "verified" as such by mathematicians today.

The small size of our checker suggests another solution. Given enough time one may read the output of the C compiler (assembly language or machine code) and verify that this output faithfully implements the C program given to the compiler. Such an examination would be tedious but it is not out of the question for a C program the size of our checker (3900 Sparc instructions, as compiled), and it could be carried out if such a high level of assurance was necessary. Such an investigation would certainly uncover Thompson-style Trojan horses inserted by a malicious compiler. This approach would not be feasible for the JVMs mentioned in the introduction; they are simply too big.

8.2 Proof-checking measurements

In order to test the proof checker, and measure its performance, we wrote a small Standard ML program that converts Twelf internal data structures into DAG format, and dumps the output of this conversion to a file, ready for consumption by the checker.

We performed our measurements on a sample proof of nontrivial size, that proves a substantial lemma that will be used in proofs of real Sparc programs. (We have not yet built a full lemma base and prover that would allow us to test full safety proofs.) In its original formulation, our sample proof is 6,367 lines of Twelf, and makes extensive use of implicit arguments. Converted to its fully explicit form, its size expands to 49,809 lines. Its DAG representation consists of 177,425 nodes.

Checking the sample proof consists of the four steps: parsing the TCB, loading the proof from disk, checking the DAG for well-formedness, and type-checking the proof itself. The first three steps take less than a second to complete, while the last step takes 79.94 seconds.

The measurements above were made on a 1 GHz Pentium III PC with 256MB of memory. During type checking of this proof the number of temporary nodes generated is 1,115,768. Most of the time during type-checking is spent in performing substitutions. All the lemmas and definitions we use in our proofs are closed expressions, and therefore they do not need to be traversed when substituting for a variable. We are currently working on an optimization that will allow our checker to keep track of closed subexpressions and to avoid their traversal during substitution. We believe this optimization can be achieved without a significant increase in the size of the checker, and it will allow us to drastically reduce type-checking time.

9 Future work

The DAG representation of proofs is quite large, and we would like to do better. One approach would be to compress the DAGs in some way; another approach is to use a compressed form of the LF syntactic notation. However, we believe that the most promising approach is neither of these.

Our proofs of program safety are structured as follows: first we prove (by hand, and check by machine) many structural lemmas about machine instructions and semantics of types [4, 5, 1]. Then we use these lemmas to prove (by hand, as derived lemmas) the rules of a low-level typed assembly language (TAL). Our TAL has several important properties:

1. Each TAL operator corresponds to exactly 0 or 1 machine instructions (0-instruction operators are coercions that serve as hints to the TAL typechecker and do nothing at runtime).

2. The TAL rules prescribe Sparc instruction encodings as well as the more conventional [17] register names, types, and so on.

3. The TAL typing rules are syntax-directed, so typechecking a TAL expression is decidable by a simple tree-walk without backtracking.

4. The TAL rules can be expressed as a set of Horn clauses.

Although we use higher-order logic to state and prove lemmas leading up to the proofs of the TAL typing rules, and we use higher-order logic in the proofs of the TAL rules, we take care to make all the statements of the TAL rules first-order Horn clauses. Consider a clause such as: `head :- goal1 , goal2 , goal3`. In LF (using our object logic) we could express this as a lemma:

```
n : pf (goal3) -> pf (goal2) ->
     pf (goal1) -> pf (head) = proof.
```

Inside *proof* there may be higher-order abstract syntax, quantification, and so on, but the `goals` are all Prolog-style. The name `n` identifies the clause.

Our compiler produces Sparc machine code using a series of typed intermediate languages, the last of which is our TAL. Our prover constructs the safety proof for the Sparc program by "executing" the TAL Horn clauses as a logic program, using the TAL expression as input data. The proof is then a tree of clause names, corresponding to the TAL typing derivation.

We can make a checker that takes smaller proofs by just implementing a simple Prolog interpreter (without backtracking, since TAL is syntax-directed). But then we would need to trust the Prolog interpreter and the Prolog program itself (all the TAL rules). This is similar to what Necula [18] and Morrisett et al. [17] do. The problem is that in a full-scale system, the TAL comprises about a thousand fairly complex rules. Necula and Morrisett have given informal (i.e., mathematical) proofs of the soundness of their type systems for prototype languages, but no

machine-checked proof, and no proof of a full-scale system.

The solution, we believe, is to use the technology we have described in this paper to check the derivations of the TAL rules from the logic axioms and the Sparc specification. Then, we can add a simple (non-backtracking) Prolog interpreter to our minimal checker, which will no longer be minimal: we estimate that this interpreter will add 200–300 lines of C code.

The proof producer (adversary) will first send to our checker, as a DAG, the definitions of Horn clauses for the TAL rules, which will be LF-typechecked. Then, the "proofs" sent for machine-language programs will be in the form of TAL expressions, which are much smaller than the proof DAGs we measured in section 8.

A further useful extension would be to implement oracle-based checking [19]. In this scheme, a stream of "oracle bits" guides the application of a set of Horn clauses, so that it would not be necessary to send the TAL expression – it would be re-derived by consulting the oracle. This would probably give the most concise safety proofs for machine-language programs, and the implementation of the oracle-stream decoder would not be too large. Again, in this solution the Horn clauses are first checked (using a proof DAG), and then they can be used for checking many successive TAL programs.

Although this approach seems very specific to our application in proof-carrying code, it probably applies in other domains as well. Our semantic approach to distributed authentication frameworks [3] takes the form of axioms in higher-order logic, which are then used to prove (as derived lemmas) first-order protocol-specific rules. While in that work we did not structure those rules as Horn clauses, more recent work in distributed authentication [12] does express security policies as sets of Horn clauses. By combining the approaches, we could have our checker first verify the soundness of a set of rules (using a DAG of higher-order logic) and then interpret these rules as a Prolog program.

10 Conclusion

Proof-carrying code has a number of technical advantages over other approaches to the security problem of mobile code. We feel that the most important of these is the fact that the trusted code of such a system can be made small. We have quantified this and have shown that in fact the trusted code can be made orders of magnitude smaller than in competing systems (JVMs). We have also analyzed the representation issues of the logical specification and shown how they relate to the size of the safety predicate and the proof checker. In our system the trusted code itself is based on a well understood and analyzed logical framework, which adds to our confidence of its correctness.

References

[1] Amal J. Ahmed, Andrew W. Appel, and Roberto Virga. A stratified semantics of general references embeddable in higher-order logic. In *In Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS 2002)*, July 2002.

[2] Andrew W. Appel. Foundational proof-carrying code. In *Symposium on Logic in Computer Science (LICS '01)*, pages 247–258. IEEE, 2001.

[3] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *6th ACM Conference on Computer and Communications Security*. ACM Press, November 1999.

[4] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *POPL '00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 243–253, New York, January 2000. ACM Press.

[5] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. on Programming Languages and Systems*, pages 657–683, September 2001.

[6] Andrew W. Appel, Neophytos G. Michael, Aaron Stump, and Roberto Virga. A Trustworthy Proof Checker. Technical Report CS-TR-648-02, Princeton University, April 2002.

[7] Andrew W. Appel and Daniel C. Wang. JVM TCB: Measurements of the trusted computing base of Java virtual machines. Technical Report CS-TR-647-02, Princeton University, April 2002.

[8] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, Henri Laulhère, César Muñoz, Chetan Murthy, Catherine Parent-Vigouroux, Patrick Loiseleur, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. The Coq Proof Assistant reference manual. Technical report, INRIA, 1998.

[9] Lujo Bauer, Michael A. Schneider, and Edward W. Felten. A general and flexible access-control system for the web. In *Proceedings of USENIX Security*, August 2002.

[10] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Ken Cline, and Mark Plesko. A certifying compiler for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*, New York, June 2000. ACM Press.

[11] Drew Dean, Edward W. Felten, Dan S. Wallach, and Dirk Balfanz. Java security: Web browers and beyond. In Dorothy E. Denning and Peter J. Denning, editors, *Internet Beseiged: Countering Cyberspace Scofflaws*. ACM Press (New York, New York), October 1997.

[12] John DeTreville. Binder, a logic-based security language. In *Proceedings of 2002 IEEE Symposium on Security and Privacy*, page (to appear), May 2002.

[13] Edward W. Felten. Personal communication, April 2002.

[14] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1979.

[15] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[16] Neophytos G. Michael and Andrew W. Appel. Machine instruction syntax and semantics in higher-order logic. In *17th International Conference on Automated Deduction*, pages 7–24, Berlin, June 2000. Springer-Verlag. LNAI 1831.

[17] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *POPL '98: 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97, New York, January 1998. ACM Press.

[18] George Necula. Proof-carrying code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, New York, January 1997. ACM Press.

[19] George C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 142–154. ACM Press, January 2001.

[20] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *The 16th International Conference on Automated Deduction*, Berlin, July 1999. Springer-Verlag.

[21] Robert Pollack. How to believe a machine-checked proof. In Sambin and Smith, editors, *Twenty Five Years of Constructive Type Theory*. Oxford University Press, 1996.

[22] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984.

Proving Cryptographic Protocols Safe From Guessing Attacks

Ernie Cohen*
Microsoft Research, Cambridge UK

June 28, 2002

Abstract

We extend the first-order protocol verification method of [1] to prove crypto protocols secure against an active adversary who can also engage in idealized offline guessing attacks. The enabling trick is to automatically construct a first-order structure that bounds the deduction steps that can appear in a guessing attack, and to use this structure to prove that such attacks preserve the secrecy invariant. We have implemented the method as an extension to the protocol verifier TAPS, producing the first mechanical proofs of security against guessing attacks in an unbounded model.

1 Introduction

Many systems implement security through a combination of strong secrets (e.g., randomly generated keys and nonces) and weak secrets (e.g. passwords and PINs). For example, many web servers authenticate users by sending passwords across SSL channels. Although most formal protocol analysis methods treat strong and weak secrets identically, it is well known that special precautions have to be taken to protect weak secrets from offline guessing attacks.

For example, consider the following simple protocol, designed to deliver an authenticated message T from a user A to a server S:

$$A \rightarrow S: \{Na, T\}_{k(A)}$$

(Here $k(A)$ is a symmetric key shared between A and S, and Na is a freshly generated random nonce.) If $k(A)$ is generated from a weak secret, an adversary might try to attack this protocol as follows:

- If A can be tricked into sending a T that is in an easily recognized sparse set (e.g., an English text, or a previously published message), the adversary can try to guess A's key. He can then confirm his guess by decrypting the message and checking that the second component is in the sparse set.
- If A can be tricked into sending the same T twice (using different nonces), or if another user B can be tricked into sending the same T, the adversary can try to guess a key for each message. He can then confirm his guess by decrypting the messages with the guessed keys, and checking that their second components are equal.

In these attacks, the adversary starts with messages he has seen and computes new messages with a sequence of *steps* (guessing, decryption, and projection in these examples). Successful attack is indicated by the discovery of an unlikely coincidence — a message (T in the examples above) that either has an unlikely property (as in the first example) or is generated in two essentially different ways (as in second example). Such a message is called a *verifier* for the attack; intuitively, a verifier confirms the likely correctness of the guesses on which its value depends.

This approach to modelling guessing attacks as a search for coincidence was proposed by Gong et. al. [2], who considered only the first kind of verifier. Lowe [3] proposed a model that includes both kinds of verifiers [1]. He also observed that to avoid false attacks, one should ignore verifiers produced by steps that simply reverse other derivation steps. For example, if an adversary guesses a symmetric key, it's not a coincidence that encrypting and then decrypting a message with this key yields the original message (it's an algebraic identity); in Lowe's formulation, such a coincidence is ignored because the decryption step "undoes" the preceding encryption step.

Lowe also described an extension to Casper/FDR that searches for such attacks by looking for a trace consisting of an ordinary protocol execution (with an active adversary), followed by a sequence of steps (dependent on a guess) that leads to a verifier. However, he did not address the problem of proving protocols safe from such attacks.

In this paper we extend the first-order verification method of [1] to prove protocols secure in the presence of an active attacker that can also engage in these kinds of offline guessing attacks. We have also extended our verifier, TAPS, to construct these proofs automatically; we believe them to be the first mechanical proofs of security against a guessing attacker in an unbounded model.

*Author's address: ernie.cohen@acm.org

[1] Lowe also considered the possibility of using a guess as a verifier; we handle this case by generating guesses with explicit attacker steps.

1.1 Roadmap

Our protocol model and verification method are fully described in [1]; in this paper, we provide only those details needed to understand the handling of guessing attacks.

We model protocols as transition systems. The state of the system is given by the set of events that have taken place (e.g., which protocol steps have been executed by which principals with which message values). We model communication by keeping track of which messages have been *published* (i.e., sent in the clear); a message is sent by publishing it, and received by checking that it's been published. The adversary is modelled by actions that combine published messages with the usual operations ((un)pairing, (en/de)cryption, etc.) and publish the result.

Similarly, a guessing attacker uses published messages to construct a guessing attack (essentially, a sequence of steps where each step contributes to a coincidence) and publishes any messages that appear in the attack (in particular, any guesses). We describe these attacks formally in section 2. Our model differs from Lowe's in minor ways (it's a little more general, and we don't allow the attack to contain redundant steps); however, we prove in the appendix that for a guesser with standard attacker capabilities, and with a minor fix to his model, the models are essentially equivalent.

To verify a protocol, we try to generate an appropriate set of first-order invariants, and prove safety properties from the invariants by first-order reasoning. Most of these invariants are invariant by construction; the sole exception is the *secrecy invariant* (described in section 4), which describes conditions necessary for the publication of a message. [1] describes how TAPS constructs and checks these invariants; in this paper, we are concerned only with how to prove that guessing attacks maintain the secrecy invariant.

In section 3, we show how to prove a bound on the information an adversary learns from a guessing attack by constructing a set of steps called a *saturation*; the main theorem says that if a set of messages has a saturation that has no verifiers, then a guessing attack starting with information from that set cannot yield information outside that set. In section 5, we show how to automatically generate first-order saturations suitable to show preservation of TAPS secrecy invariants. TAPS uses a resolution theorem prover to check that the saturation it constructs really does define a saturation, and to check that this saturation has no verifiers.

2 Guessing Attacks

In this section, we describe guessing attacks abstractly. We also define a *standard model*, where the attacker has the usual Dolev-Yao adversary capabilities (along with the ability to guess and recognize).

We assume an underlying set of *messages*; variables X, Y, Z, A, Na, and T range over messages, and M over arbitrary sets of messages. In the standard model, the message space comes equipped with the following functions (each injective[2], with disjoint ranges):

nil	a trivial message
$\{X, Y\}$	the ordered pair formed from X and Y
X_Y	encryption of X under the key Y

Following Lowe, we define a guessing attacker by the *steps* that he can use to construct an attack; variables starting with s denote steps, variables starting with S denote sets of steps or finite sequences of steps without repetition (treating such sequences as totally ordered sets). We assume that each step has a finite sequence of message *inputs* and a single message *output*; intuitively, a step models an operation available to the adversary that, produces the output from the inputs. An input/output of a set of steps is an input/output of any of its members. We say s is *inside* M iff the inputs and output of s are all in M, and is *outside* M otherwise. S is inside M iff all its steps are inside M.

In the standard model, a step is given by a vector of length three, the first and third elements giving its inputs and output, respectively. Steps are of the following forms, where d,$guessable$, and $checkable$ are protocol-dependent predicates described below:

$\langle\langle\rangle, nil, nil\rangle$
$\langle\langle X, Y\rangle, cons, \{X, Y\}\rangle$
$\langle\langle X, Y\rangle, enc, X_Y\rangle$
$\langle\langle\{X, Y\}\rangle, car, X\rangle$
$\langle\langle\{X, Y\}\rangle, cdr, Y\rangle$
$\langle\langle X_Y, Z\rangle, dec, X\rangle$, where $d(Y, Z)$
$\langle\langle\rangle, guess, X\rangle$ where $guessable(X)$
$\langle\langle X\rangle, check, nil\rangle$ where $checkable(X)$

The first five steps model the adversary's ability to produce the empty message, to pair, encrypt and project. The sixth step models his ability to decrypt; $d(X, Y)$ means that messages encrypted under X can be decrypted using Y[3]. The seventh step models the adversary's ability to guess; we think of the adversary as having a long list of likely secrets, and $guessable(X)$ just means that X is in the list. The last step models the adversary's ability to recognize members of particular sparse sets (such as English texts); $checkable(X)$ means X passes the recognition test[4].

[2] As usual, we cheat in assuming pairing to be injective; to avoid a guesser being able to recognize pairs, pairing is normally implemented as concatenation, which is not generally injective. One way to improve the treatment would be to make message lengths explicit.

[3] For example, the axiom $sk(X) \Leftrightarrow (\forall Y : d(X, Y) \Leftrightarrow X = Y)$ defines sk as a recognizer for symmetric keys.

[4] Because of the way we define attacks, $checkable$ has to be defined with some care. For example, defining $checkable$ to be all messages of the form $\{X, X\}$ would allow construction of a verifier for any guess. However, we can use $checkable$ to recognize asymmetric key pairs, which were treated by Lowe as a special case.

Let *undoes* be a binary relation on steps, such that if $s1$ undoes $s2$, then the output of $s1$ is an input of $s2$ and the output of $s2$ is an input of $s1$[5]. Intuitively, declaring that $s1$ undoes $s2$ says that performing $s1$ provides no new "information" if we've already performed $s2$[6]. Typically, data constructors and their corresponding destructors are defined to undo each other.

In the standard model, we define that step $s1$ undoes step $s2$ iff there are X, Y, Z such that one of the following cases holds:

1. $s1 = \langle\langle X, Y\rangle, cons, \{X, Y\}\rangle$
   $s2 = \langle\langle\{X, Y\}\rangle, car, X\rangle$
2. $s1 = \langle\langle X, Y\rangle, cons, \{X, Y\}\rangle$
   $s2 = \langle\langle\{X, Y\}\rangle, cdr, Y\rangle$
3. $s1 = \langle\langle X, Y\rangle, enc, X_Y\rangle$
   $s2 = \langle\langle X_Y, Z\rangle, dec, X\rangle$
4. any of the previous cases with $s1$ and $s2$ reversed

A sequence of steps S is a *pre-attack* on M if (1) every input to every step of S is either in M or the output of an earlier step of S, and (2) no step of S undoes an earlier step of S. An *M-verifier* of a set of steps S is the output of a step of S that is in M or the output of another step of S. A pre-attack on M, S, is an *attack* on M iff the output of every step of S is either an M-verifier of S or the input of a later step of S. Intuitively, an attack on M represents an adversary execution, starting from M, such that every step generates new information, and every step contributes to a coincidence (i.e., a path to a verifier). Note that attacks are monotonic: if $M \subseteq M'$, an attack on M is also an attack on M'. An attack on M is *effective* iff it contains a step outside M. Note also that removing a step inside M from an (effective) attack on M leaves an (effective) attack on M.

A guessing attacker is one who, in addition to any other capabilities, can publish any messages that appear as inputs outputs in an attack on the set of published messages[7].

example: We illustrate with some examples taken from [3]. In each case, we assume g is guessable.

- If $g_g \in M$, then

  $$\langle\langle\rangle, guess, g\rangle, \langle\langle g, g\rangle, enc, g_g\rangle$$

  is an attack on M
- If $g_g \in M$ and g is a symmetric key, then

  $$\langle\langle\rangle, guess, g\rangle, \langle\langle g_g, g\rangle, dec, g\rangle$$

  is an attack on M
- If $M = \emptyset$ and g is a symmetric key, then

  $$\langle\langle\rangle, guess, g\rangle, \langle\langle g, g\rangle, enc, g_g\rangle, \langle\langle g_g, g\rangle, dec, g\rangle$$

  is not an attack, or even a pre-attack, on M, because the last step undoes the second step.
- If $\{v, v\}_g \in M$, where g is a symmetric key,

  $$\langle\langle\rangle, guess, g\rangle, \langle\langle\{v, v\}_g, g\rangle, dec, \{v, v\}\rangle,$$
  $$\langle\langle\{v, v\}\rangle, car, v\rangle, \langle\langle\{v, v\}\rangle, cdr, v\rangle$$

  is an attack on M.

If M in the last example is the set of published messages, the attack has the effect of publishing the messages $g, \{v, v\}_g, \{v, v\}$, and v.
end of example

3 Saturation

How can we prove that there are no effective attacks on a set of messages M? One way is to collect together all messages that might be part of an effective attack on M, and show that this set has no verifiers. However, this set is inductively defined and therefore hard to reason about automatically. We would rather work with a first-order approximation to this set (called a saturation below); this approximation might have additional steps (e.g., a set of steps forming a loop, or steps with an infinite chain of predecessors); nevertheless, as long as it has no verifiers, it will suit our purposes.

However, there is a catch; to avoid false attacks, we don't want to add a step to the set if it undoes a step already in the set. Thus, adding extra steps to the set might cause us to not include other steps that should be in the set (and might be part of an effective guessing attack). Fortunately, it turns out that this can happen only if the set has a verifier.

Formally, a set of steps S *saturates* M iff every step outside M whose inputs are all in $M \cup outputs(S)$ is either in S or undoes some step in S. The following theorem says that we can prove M free from effective attacks by constructing a saturation of M without an M-verifier, and that this method is complete:

[5]The peculiar requirement on undoing has no real intuitive significance, but is chosen to make the proof of the saturation theorem work. The requirement that the output of $s1$ is an input of $s2$ is not strictly necessary, but it makes the proof of completeness part of the theorem easier.

[6]One of the weaknesses of our model (and Lowe's) is that the $undoes$ relation does not distinguish between it's inputs. For example, encryption with a public key followed by decryption with a private key does not reveal any new information about the plaintext, but does reveal information about the keys (namely, that they form a key pair). This is why asymmetric key pairs have to be handled with $check$ steps.

[7]Because the set of published messages is generally not closed under adversary actions, an attack might publish new messages even without using guesses. However, such any message published through such an attack could equally be published through ordinary adversary actions.

Theorem 1 *There is an effective attack on M iff every saturation of M has an M-verifier.*

To prove the forward implication, let A be an effective attack on M, and let S saturate M; we show that S has an M-verifier. Since A is effective, assume wlog that A has no steps inside M. If $A \subseteq S$, the M-verifier for A is also an M-verifier for S. Otherwise, let a be the first step of A not in S; all inputs of a are in outputs of previous steps of A (which are also steps of S) or in M. Since S saturates M and $a \notin S$, a undoes some step $s \in S$; by the condition on undoing, the output x of s is an input of a. Moreover, because A is a pre-attack, x is either in M or the output of an earlier step $a1$ of A (hence $a1 \in S$) that a does not undo (and hence, since a undoes s, $a1 \neq s$). In either case, x is an M-verifier for S.

To prove the converse, let S be the set of all steps of all pre-attacks on M whose steps are outside M; we show that (1) S saturates M and (2) if S has a verifier, there is an effective attack on M. To show (1), let s be a step outside M whose inputs are all in $S \cup M$ and does not undo a step of S; we have to show that $s \in S$. Since each input of s that is not in M is the output of a step of a pre-attack on M, concatenate these pre-attacks together (since s has only finitely many inputs) and repeatedly delete steps that undo earlier steps (by the condition on the $undoes$ relation, such deletions do not change the set of messages that appear in the sequence); the result is a pre-attack on M. Since s does not undo any step of S, it does not undo any steps of this sequence, so adding it to the end gives a pre-attack on M; thus $s \in S$. To show (2), suppose S has an M-verifier x; then there are steps $s1, s2 \in S$ such that x is the output of $s1$ and $s2$ and either $x \in M$ or $s1 \neq s2$. By the definition of S, $s1$ and $s2$ are elements of pre-attacks A and B with steps outside M, and x is an M-verifier for $A \cup B$. Thus, $A; B$ is a pre-attack on M with a verifier; by repeatedly removing from this pre-attack steps that produce outputs that are neither M-verifiers nor inputs to later steps, we are eventually left with an effective attack on M.

4 The secrecy invariant

In our verification method (for nonguessing attackers), the secrecy invariant has the form

$$pub(X) \Rightarrow ok(X)$$

where $pub(X)$ is a state predicate that means X has been published and ok is the strongest predicate[8] satisfying the following formulas:

$$\begin{array}{rcl} prime(X) & \Rightarrow & ok(X) \\ & & ok(nil) \\ pub(X) \wedge pub(Y) & \Rightarrow & ok(\{X, Y\}) \\ pub(X) \wedge pub(Y) & \Rightarrow & ok(X_Y) \end{array}$$

The computation of an appropriate definition of the predicate $prime$ and the proof obligations needed to show the invariance of the secrecy invariant are fully described in [1]; for our purposes, the relevant properties of $prime$ are

$$\begin{array}{rcl} prime(\{X, Y\}) & \Rightarrow & prime(X) \wedge prime(Y) \\ prime(X_Y) \wedge dk(Y) & \Rightarrow & prime(X) \end{array}$$

where $dk(X) \Leftrightarrow (\exists\ Y : d(X, Y) \wedge pub(Y))$. Intuitively, $dk(X)$ means that the adversary possesses a key to decrypt messages encrypted under X.

example (continued): Consider the protocol described in the introduction. To simplify the example, we assume that $k(A)$ is unpublished, for all A, and that Na and T are freshly generated every time the protocol is run. For this protocol, TAPS defines $prime$ to be the strongest predicate satisfying

$$(1) \qquad p0(A, Na, T) \quad \Rightarrow \quad prime(\{Na, T\}_{k(A)})$$

where $p0(A, Na, T)$ is a state predicate that records that A has executed the first step of the protocol with nonce Na and message T. (Note that if some keys could be published, the definition would include additional primality cases for $\{Na, T\}$, Na, and T.)
end of example

To prove that the secrecy invariant is preserved by guessing steps, we want to prove that any value published by a guessing attack is ok when the attacker publishes it. Inductively, we can assume that the secrecy invariant holds when a guessing attack is launched, so we can assume that every published message is ok. Because attacks are monotonic in the starting set of messages, it suffices to show that there are no effective attacks on the set of ok messages; by the saturation theorem, we can do this by constructing a saturation without verifiers for the set of ok messages.

5 Generating a Saturation

We define our saturation as follows; in each case, we eliminate steps inside ok to keep the saturation as small as possible. We say a message is *available* if it is ok or is the output of a step of the saturation. We call car,cdr, and dec steps *destructor* steps, and the remaining steps *constructor* steps. The *main input* of a destructor step is the encrypted message in the case of a dec step and the sole input in the case of a car or cdr step. A constructor step outside ok is

[8] Because ok does not occur in the definition of $prime$, we could instead define ok as an ordinary disjunction, but it's more convenient to work with sets of formulas than big disjunctions. The same remarks apply to the predicates $guessable$, $checkable$, cc, scc, sd, and $coreOK$ below.

in the saturation iff its inputs are available and it does not undo a destructor step of the saturation. A destructor step outside ok is in the saturation if it is a *core* step (defined below); we define the core so that it includes any destructor step outside ok whose inputs are available and whose main input is either $prime$, the output of a guessing step, or the output of a core step.

To see that this definition is consistent, note that if we take the same definition but include in the core all constructor steps outside ok whose inputs are available (i.e. ignoring whether they undo destructor steps), the conditions on the saturation and availability form a Horn theory, and so have a strongest solution. Removing from the saturation those constructor steps that undo core steps doesn't change the definition of availability or the core (since the main input to every core step is either $prime$ or the output of a core step).

To show that this definition yields a saturation of ok, we have to show that any step outside ok whose inputs are available is either in the saturation or undoes a step of the saturation. This is immediate for constructor steps and for destructor steps whose main input is $prime$, a guess, or the output of a core step. A destructor step whose main input is the output of a constructor step in the saturation (other than a $guess$) undoes the constructor step (because encryption and pairing are injective with disjoint ranges). Finally, for a destructor step whose main input is ok but not $prime$, the main input must be a pair or encryption whose arguments are published (and hence ok by the secrecy invariant), so such a destructor step is inside ok.

Finally, we need to define the core. Let $sd(X_Y)$ mean that $\langle\{X_Y, Z\}, dec, X\rangle$ is a core step for some Z[9], let $scc(\{X,Y\})$ mean that $\langle\langle\{X,Y\}\rangle, car, X\rangle$ and $\langle\langle\{X,Y\}\rangle, cdr, Y\rangle$ are core steps[10], and let $cc(X)$ ("X is a core candidate") mean that X is either prime, a guess, or the output of a core step. For the remainder of the description, we assume that guessability and primality are given as the strongest predicates satisfying a finite set of equations of the forms $\mathsf{f} \Rightarrow prime(\mathsf{X})$ and $\mathsf{f} \Rightarrow guessable(\mathsf{X})$. Let $\mathcal{F}$ be the set of formulas $\mathsf{f} \Rightarrow cc(\mathsf{X})$, where $\mathsf{f} \Rightarrow prime(\mathsf{X})$ is a formula defining $prime$ or $\mathsf{f} \Rightarrow guessable(\mathsf{X})$ is a formula defining $guessable$. These formulas guarantee that cc includes all prime messages and all guessable messages.

example (continued): For our example, we assume that all keys are guessable, so g is defined as the strongest predicate satisfying

(2) $\quad guessable(k(A))$

[9]We don't need to keep track of what decryption key was used (when creating the saturation), because the $undoes$ relation for decryptions is independent of decryption key. However, when we check for verifiers, we have to make sure that at most one decryption key is available for each of these encryptions.

[10]For the saturation defined here, each of these steps are in the core iff $cc(\{X,Y\})$

Since $prime$ and $guessable$ are defined by formulas (1) and (2) respectively, $\mathcal{F}$ initially contains the formulas

(3) $\quad p0(A, Na, T) \Rightarrow cc(\{Na, T\}_{k(A)})$

(4) $\quad cc(k(A))$

end of example

To make sure that $\mathcal{F}$ defines a suitable core (as defined above), we expand $\mathcal{F}$ to additionally satisfy the formulas

(5) $\quad sd(X_Y) \Rightarrow cc(X)$

(6) $\quad scc(\{X,Y\}) \Rightarrow cc(X)$

(7) $\quad scc(\{X,Y\}) \Rightarrow cc(Y)$

(8) $\quad cc(\{X,Y\}) \wedge sc(\{X,Y\}) \Rightarrow scc(\{X,Y\})$

(9) $\quad cc(X_Y) \wedge se(X_Y) \Rightarrow sd(X_Y)$

where

$$se(X_Y) \Leftrightarrow (\exists\, Z : d(Y,Z) \wedge avail(Z) \wedge (\neg ok(X_Y) \vee \neg ok(Z) \vee \neg ok(X)))$$
$$sc(\{X,Y\}) \Leftrightarrow \neg ok(\{X,Y\})$$

(5)-(7) say that cc includes all outputs of core steps; (8) says that a car or cdr step is in the core if it is outside ok and its input is in cc; and (9) says that a decryption step is in the core if it is outside ok and a decryption key is available. To construct the formulas for sd, scc, and cc meeting these conditions, we repeatedly add formulas to $\mathcal{F}$ as follows:

- If $(\mathsf{f} \Rightarrow cc(\mathsf{X_Y})) \in \mathcal{F}$, add to $\mathcal{F}$ the formulas

$$\mathsf{f} \wedge se(\mathsf{X_Y}) \Rightarrow sd(\mathsf{X_Y})$$
$$\mathsf{f} \wedge se(\mathsf{X_Y}) \Rightarrow cc(\mathsf{X})$$

example (continued): Applying this rule to the formula (3), we add to $\mathcal{F}$ the formulas

$$p0(A, Na, T) \wedge se(\{Na, T\}_{k(A)}) \Rightarrow sd(\{Na, T\}_{k(A)})$$
$$p0(A, Na, T) \wedge se(\{Na, T\}_{k(A)}) \Rightarrow cc(\{Na, T\})$$

Since $k(A)$ is a symmetric key, $d(k(A), V)$ simplifies to $V = k(A)$, and both $avail(k(A))$ and $se(\{Na, T\}_{k(A)})$ simplify to $true$, so we can simplify these formulas to

(10) $\quad p0(A, Na, T) \Rightarrow sd(\{Na, T\}_{k(A)})$

(11) $\quad p0(A, Na, T) \Rightarrow cc(\{Na, T\})$

end of example

- If $(\mathsf{f} \Rightarrow cc(\{\mathsf{X},\mathsf{Y}\})) \in \mathcal{F}$, add to $\mathcal{F}$ the formulas

$$\begin{array}{lcl} \mathsf{f} \wedge sc(\{\mathsf{X},\mathsf{Y}\}) & \Rightarrow & scc(\{\mathsf{X},\mathsf{Y}\}) \\ \mathsf{f} \wedge sc(\{\mathsf{X},\mathsf{Y}\}) & \Rightarrow & cc(\mathsf{X}) \\ \mathsf{f} \wedge sc(\{\mathsf{X},\mathsf{Y}\}) & \Rightarrow & cc(\mathsf{Y}) \end{array}$$

example (continued): Applying this rule to (11), and using the fact that with the given secrecy invariant, $p0(A, Na, T) \Rightarrow \neg ok(\{Na, T\})$, yields the formulas

$$\begin{array}{lcl} p0(A, Na, T) & \Rightarrow & scc(\{Na, T\}) \\ p0(A, Na, T) & \Rightarrow & cc(Na) \\ p0(A, Na, T) & \Rightarrow & cc(T) \end{array} \quad (12)\ (13)\ (14)$$

end of example

- If $(\mathsf{f} \Rightarrow cc(\mathsf{X})) \in \mathcal{F}$, where X is a variable symbol or an application of a function other than $cons$ or enc, add to $\mathcal{F}$ the formula

$$\mathsf{f} \Rightarrow coreOK(\mathsf{X})$$

where $coreOK$ is the strongest predicate such that

$$\begin{array}{lcl} coreOK(X) & \Leftrightarrow & (sc(X) \Rightarrow scc(X)) \\ & & \wedge (se(X) \Rightarrow sd(X)) \end{array}$$

Intuitively, $coreOK(X)$ means that we believe that we don't need to generate any more core cases to handle X, because we think that f implies that if X is the main input of a destructor step, then it is already handled by one of the other cases in $\mathcal{F}$. (Typically, it is because we think that the authentication properties of the protocol guarantee that X will not be an encryption or pair[11].) This bold assumption is just a good heuristic[12]; if it turns out to be wrong, TAPS is unable to prove the resulting proof obligation, and the whole protocol proof fails (i.e., the assumption does compromise soundness).

example (continued): Applying this rule to (13),(14), and (4) yields the obligations

$$\begin{array}{lcl} p0(A, Na, T) & \Rightarrow & coreOK(Na) \\ p0(A, Na, T) & \Rightarrow & coreOK(T) \\ & & coreOK(k(A)) \end{array} \quad (15)\ (16)\ (17)$$

end of example

After this process reaches a fixed point[13], we define sd to be the strongest predicate satisfying the formulas of $\mathcal{F}$ of the form $\mathsf{f} \Rightarrow sd(\mathsf{X})$, and similarly for scc and cc.

example (continued): The formulas of $\mathcal{F}$ generate the definitions

$$\begin{array}{rcl} sd(\{Na, T\}_{k(A)}) & \Leftrightarrow & p0(A, Na, T) \\ scc(\{Na, T\}) & \Leftrightarrow & p0(A, Na, T) \end{array}$$

end of example

These definitions guarantee that the core satisfies (5)-(7); to show that it satisfies (8)-(9), we also need to prove explicitly the remaining formulas from $\mathcal{F}$ of the form $\mathsf{f} \Rightarrow coreOK(X)$. TAPS delegates these proofs to a resolution prover; if these proofs succeed, we have successfully defined the core, and hence, the saturation[14].

example (continued): The remaining obligations are (15)-(16) and (17). (15)-(16) follow trivially from the fact that $p0(A, Na, T)$ implies that A,Na, and T are all atoms; (17) depends on a suitable axiom defining the keying function k (e.g., that $k(A)$ is an atom, or is an encryption under a hash function).
end of example

Finally, we have to prove that the saturation has no verifiers. Because of the way in which our saturation is constructed, we can eliminate most of the cases as follows. The output of a $cons$ step cannot be the output of any other step of the saturation —

- it can't be the output of a core step (because the core would include the car and cdr steps generated from the output, which would be undone by the $cons$ step);
- it can't be the output of another $cons$ step (because pairing is injective) or an enc step (since $cons$ and enc have disjoint ranges);
- it can't be ok, because both of its inputs would also be published (by the definition of ok), hence ok, and so the step would be inside ok (hence out of the saturation).

Similarly, the output of an enc step cannot be the output of another enc step (because enc is injective).

Thus, we can prove the saturation has no verifiers by showing that each of the remaining cases is impossible:

[11] Unfortunately, a proof that shows this interplay between authentication reasoning a safety from guessing would be too large to present here; analogous proofs that show the synergy between authentication and secrecy reasoning are given in [1].

[12] Like all such choices in TAPS, the user can override this choice with hints about how to construct the saturation.

[13] The process reaches a fixed point by induction on the size of the term on the right of the implication; it does not require an induction principle on messages.

[14] In fact, the way the TAPS secrecy invariant is constructed, these proofs are probably not necessary. But they also present no problems if TAPS can prove the secrecy invariant (we have never seen one fail), so we have left them in for the sake of safety.

- an enc step whose output v is $prime$ or the output of a core step, such that no decryption key for v is available (This case is trivial for symmetric encryptions, since the key (and hence also a decryption key) is available.)
- two distinct core steps with the same output
- a core step whose output is ok or guessable
- a saturation step whose output is checkable

These cases are relatively easy to check, because they involve messages of known structure and pedigree.

example (continued): To prove that this saturation has no verifiers, we have to check the following cases:

- the output of a enc step that is $prime$ or the output of a core step, without an available decryption key: this case goes away because the only such encryptions are symmetric.
- a core step that produces an ok value; this case checks because the secrecy invariant guarantees that neither Na nor T is ok.
- a core step that produces a guessable value; this depends on an additional assumptions about k (e.g., $k(A)$ is an atom, but not a value generated for Na or T[15]).
- two distinct core steps that produce the same value. A dec step cannot collide with a car or cdr step - the former produces $cons$ terms, while the latter produce atoms. Two core dec steps cannot collide because Na is freshly chosen and thus uniquely determines A and T (and thus, by the injectivity properties of $cons$, $\{Na, T\}$ uniquely determines both $\{Na, T\}_{k(A)}$ and $k(A)$; because $k(A)$ is symmetric, it also uniquely determines the decryption key). Two car (respectively, cdr) steps cannot collide because Na (respectively, T) is freshly chosen, and hence functionally determines $\{Na, T\}$. Finally, a car and a cdr step cannot collide because the same value is not chosen as both an Na value and an T value. Note that these cases would fail if the same value could be used for Na and T, or if an Na or T value could be reused.
- a saturation step whose output is checkable; in this example, this is trivial, since no messages are regarded as checkable. Were we not able to prove directly that T is uncheckable, the proof would fail.

end of example

[15] In practice, we avoid having to write silly axioms like this by generating keys like $k(A)$ with protocol actions; nonce unicity lemmas [1] then give us all these conditions for free.

In general, the total number of proof obligations grows roughly as the square of the number of core cases, in contrast to the ordinary protocol reasoning performed by TAPS, where the number of obligations is roughly linear in the number of prime cases. Moreover, many of the proofs depend on authentication properties[16] Thus, as expected, the guessing part of the verification is typically much more work than the proof of the secrecy invariant itself. For example, for EKE, TAPS proves the secrecy invariant and the authentication properties of the protocol in about half a second, while checking for guessing attacks takes about 15 seconds[17]. However, the proofs are completely automatic.

While we have applied the guessing attack extension to TAPS to verify a number of toy protocols (and a few less trivial protocols, like EKE), a severe limitation is that TAPS (like other existing unbounded crypto verifiers) cannot handle Vernam encryption. The next major revision of TAPS will remedy this situation, and allow us to more reasonably judge the usefulness of our approach to guessing.

6 Acknowledgements

This work was done while visiting Microsoft Research, Cambridge; we thank Roger Needham and Larry Paulson for providing the opportunity. We also thank Gavin Lowe for patiently fielding many stupid questions about guessing attacks, and the referees for their insightful comments.

References

[1] E. Cohen, *First-Order Verification of Cryptographic Protocols.* In *JCS* (to appear). A preliminary version appears in *CSFW* (2000)

[2] L. Gong and T. Mark and A. Lomas and R. Needham and J. Saltzer, *Protecting Poorly Chosen Secrets from Guessing Attacks.* IEEE Journal on Selected Areas in Communications, 11(5):648-656 (1993)

[3] G. Lowe, *Analyzing Protocols Subject to Guessing Attacks.* In *WITS* (2002)

[4] C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, T. Engel, E. Keen, C. Theobalt, and D. Topić, *System description: SPASS version 1.0.0.* In *CADE 15*, pages 378–382 (1999).

[16] For example, if a protocol step publishes an encryption with a component received in a message, and the message is decrypted in a guessing attack, showing that the destructor step that extracts the component doesn't collide depends on knowing where the component came from.

[17] This is in part due to the immaturity of the implementation; it takes some time to find the right blend of theorem proving and preprocessing to optimize performance. For example, the current TAPS implementation effectively forces the prover into splitting the proof into cases, since the prover is otherwise reluctant to do resolutions that introduce big disjunctions.

A Lowe's Guessing Model

In this appendix, we describe Lowe's model for guessing attacks, and sketch a proof that for the standard model and with a single guess, Lowe's model and ours are essentially equivalent.

Throughout this section, we work in the standard model. Moreover, to match Lowe's model, we define the checkable messages to be all pairs of asymmetric keys, and define a single value g to be guessable. Let M be a fixed set of messages. An *execution* is a finite sequence of distinct steps such that every input to every step is either in M or the output of an earlier step of S[18]. A message m is *new* iff every execution from M that contains m as an input or output also includes a guessing step; intuitively, new messages cannot be produced from M without the use of g. A *Lowe attack* is an execution with steps $s1$ and $s2$ (not necessarily distinct), each with output v, satisfying the following properties:

- $s1$ is either a guessing step or has an input that is new (Lowe's condition (3));
- either (1) $s1 \neq s2$, (2) v is in M, or (3) v is an asymmetric key and v^{-1} is in M or the output of a step of the execution (Lowe's condition (4))
- neither $s1$ nor $s2$ undoes any step of the execution (Lowe's condition (5))

Theorem 2 *There is a Lowe attack on M iff there is an attack on M that reveals g.*

First, let S be a Lowe attack with $s1$, $s2$, and v as in the definition above; we show how to turn S into an attack on M that reveals g. Let $s3$ be a step other than $s1$ or $s2$. If the output of $s3$ is not an input of a later step, then deleting $s3$ from S leaves a Lowe attack revealing. If $s3$ undoes an earlier step of S, then (in the standard model) the output of $s3$ is an input of the earlier step, so again deleting $s3$ from S leaves a Lowe attack revealing g. Repeating these deletions until no more are possible leaves a Lowe attack revealing g where no step undoes an earlier step, and where the output of every step other than $s1$ and $s2$ is an input to a later step.

If $s1 \neq s2$, or if $s1 = s2$ and the output of $s1$ is in M or the output of another step of S, then the outputs of $s1$ and $s2$ are verifiers and S is an attack. Otherwise, $s1 = s2$, v is an asymmetric key, and v^{-1} is in M or the output of a step of S. If v is an input to a step that follows $s1$, then again S is an attack. If $\{v, v^{-1}\}$ is neither in M nor the output of a step of G, then add to the end of S the steps $\langle\langle v, v^{-1}\rangle, cons, \{v, v^{-1}\}\rangle, \langle\langle\{v, v^{-1}\}\rangle, check, nil\rangle$ to create an attack. Otherwise, let $s3 = \langle\langle\{v, v^{-1}\}\rangle, car, v\rangle$. If $s1 = s3$, delete $s1$ from S and add to the end of S the step $\langle\{v, v^{-1}\}\rangle, check, nil\rangle$; if $s1 \neq s3$, add $s3$ to the end of S. In either case, the resulting S is an attack.

Conversely, let S be an attack on M that reveals g. Note that since S is a pre-attack, no step of S undoes a previous step of S; since the undoing relation is symmetric in the standard model, no step of S undoes any other step of S. Since g is not derivable from M, it must be the output of a guessing step of S; without loss of generality, assume it is the output of the first step.

- If g is a verifier for S, then since g is not in M, it must also be the output of another step $s1$ of S. Because S is an attack, $s1$ does not undo any earlier step of S. Thus, the prefix of S up to and including $s1$ is a Lowe attack on M.
- If g is not a verifier for S, then g must be an input to a later step of S. Because g is new, some step of S has a new input; since S is finite, there is a step s of S with a new input such that no later step has a new input. Thus, the output of s is either not new or is a verifier of S. If the output of s is not new and not a verifier of S, append to S an execution without guessing or checking steps that outputs the output of s, deleting any added steps that undo steps of S. This produces an attack with a step whose output is a verifier for S and has an input that is new. If s is not a checking step, then S is a Lowe attack. If s is a checking step, then one input to s is an asymmetric key k that is new, such that k^{-1} is either in M or the output a step that precedes s. Since k is new, it must be the output of a step $s2$ of S that is either a guessing step or has a new input; in either case, S is a Lowe attack, with $s2$ in the role of $s1$.

[18] Lowe actually did not have guessing steps, but instead simply allowed g to appear as inputs to steps of S and as a verifier. However, because guesses did not appear as explicit steps, he missed the following case: suppose g is an asymmetric key and M is the set $\langle g^{-1}\rangle$. Obviously this should be considered to be vulnerable to a guessing attack. However, there is no attack in Lowe's original model, because any *step* producing g or g^{-1} would have to undo a previous step.

Automatic SAT-Compilation of Protocol Insecurity Problems via Reduction to Planning*

Alessandro Armando† and Luca Compagna†

June 28, 2002

Abstract

We provide a fully automatic translation from security protocol specifications into propositional logic which can be effectively used to find attacks to protocols. Our approach results from the combination of a reduction of protocol insecurity problems to planning problems and well-known SAT-reduction techniques developed for planning. We also propose and discuss a set of transformations on protocol insecurity problems whose application has a dramatic effect on the size of the propositional encoding obtained with our SAT-compilation technique. We describe a model-checker for security protocols based on our ideas and show that attacks to a set of well-known authentication protocols are quickly found by state-of-the-art SAT solvers.

Keywords: Foundation of verification; Confidentiality and authentication; Intrusion detection.

1 Introduction

Even under the assumption of perfect cryptography, the design of security protocols is notoriously error-prone. As a consequence, a variety of different protocol analysis techniques has been put forward [3, 4, 8, 10, 12, 16, 19, 22, 23]. In this paper we address the problem of translating protocol insecurity problems into propositional logic in a fully automatic way with the ultimate goal to build an automatic model-checker for security protocols based on state-of-the-art SAT solvers. Our approach combines a reduction of protocol insecurity problems to planning problems[1] with well-known SAT-reduction techniques developed for planning. At the core of our technique is a set of transformations whose application to the input protocol insecurity problem has a dramatic effect on the size of the propositional formulae obtained. We present a model-checker for security protocols based on our ideas and show that—using our tool—attacks to a set of well-known authentication protocols are quickly found by state-of-the-art SAT solvers.

2 Security Protocols and Protocol Insecurity Problems

In this paper we concentrate our attention on error detection of authentication protocols (see [7] for a survey). As a simple example consider the following one-way authentication protocol:

$$(1)\ \ A \rightarrow B : \{N_a\}_{K_{ab}}$$
$$(2)\ \ B \rightarrow A : \{f(N_a)\}_{K_{ab}}$$

where N_a is a nonce[2] generated by Alice, K_{ab} is a symmetric key, f is a function known to Alice and Bob, and $\{x\}_k$ denotes the result of encrypting text x with key k. Successful execution of the protocol should convince Alice that she has been talking with Bob, since only Bob could have formed the appropriate response to the message issued in (1). In fact, Ivory can deceit Alice into believing that she is talking with Bob whereas she is talking with her. This is achieved by executing concurrently two sessions of the protocol and using messages from one session to form messages in the other as illustrated by the following protocol trace:

$$(1.1)\ \ A \rightarrow I(B) : \{N_a\}_{K_{ab}}$$
$$(2.1)\ \ I(B) \rightarrow A : \{N_a\}_{K_{ab}}$$
$$(2.2)\ \ A \rightarrow I(B) : \{f(N_a)\}_{K_{ab}}$$
$$(1.2)\ \ I(B) \rightarrow A : \{f(N_a)\}_{K_{ab}}$$

Alice starts the protocol with message (1.1). Ivory intercepts the message and (pretending to be Bob) starts a second session with Alice by replaying the received message—cf. step (2.1). Alice replies to this message with message (2.2). But this is exactly the message Alice

*This work has been supported by the Information Society Technologies Programme, FET Open Assessment Project "AVISS" (Automated Verification of Infinite State Systems), IST-2000-26410.

†DIST – Università degli Studi di Genova, Viale Causa 13 – 16145 Genova, Italy, {armando,compa}@dist.unige.it

[1] The idea of regarding security protocol analysis as a planning problem is not new. To our knowledge it is also been proposed in [1].

[2] *Nonces* are numbers generated by principals that are intended to be used *only once*.

is waiting to receive in the first protocol session. This allows Ivory to finish the first session by using it—cf. (*1.2*). At the end of the above steps Alice believes she has been talking with Bob, but this is obviously not the case.

A problem with the above rule-based notation to specify security protocols is that it leaves implicit many important details such as the shared information and how the principals should react to messages of an unexpected form. This kind of description is therefore usually supplemented with explanations in natural language which in our case explain that N_a is a nonce generated by Alice, that f is a function known to the honest participants, and that K_{ab} is a shared key.

To cope with the above difficulties and pave the way to the formal analysis of security protocols a set of models and specification formalisms as well as translators from high-level languages (similar to the one we used above to introduce our example) into these formalisms have been put forward. For instance, Casper [18] compiles high-level specifications into CSP, whereas CAPSL [5] and the AVISS tool [2] compile high-level specifications into formalisms based on multiset rewriting inspired by [6].

2.1 The Model

We model the concurrent execution of a protocol by means of a state transition system. Following [16], we represent states by sets of atomic formulae called *facts* and transitions by means of rewrite rules over sets of facts. For the simple protocol above, facts are built out of a first-order sorted signature with sorts `user`, `number`, `key`, `func`, `text` (super-sort of all the previous sorts), `int`, `session`, `nonceid`, and `list_of text`. The constants 0, 1, and 2 (of sort `int`) denote protocols steps, $\underline{1}$ and $\underline{2}$ (of sort `session`) denote session instances, a and b (of sort `user`) denote honest participants, k_{ab} (of sort `key`) denotes a symmetric key and na (of sort `nonceid`) is a nonce identifier. The function symbol $\{\_\}\_ : \texttt{text} \times \texttt{key} \rightarrow \texttt{text}$ denotes the encryption function, $f : \texttt{number} \rightarrow \texttt{func}$ denotes the function known to the honest participants, $nc : \texttt{nonceid} \times \texttt{session} \rightarrow \texttt{number}$, and $s : \texttt{session} \rightarrow \texttt{session}$ are nonce and session constructors respectively. The predicate symbols are i of arity `text`, *fresh* of arity `number`, m of arity $\texttt{int} \times \texttt{user} \times \texttt{user} \times \texttt{text}$, and w of arity $\texttt{int} \times \texttt{user} \times \texttt{user} \times \texttt{list\_of text} \times \texttt{list\_of text} \times \texttt{session}$:

- $i(t)$ means that the intruder knows t.
- *fresh*(n) means that n has not been used yet.
- $m(j, s, r, t)$ means that principal s has (supposedly)[3] sent message t to principal r at protocol step j.
- $w(j, s, r, ak, ik, c)$ represents the state of execution of principal r at step j of session c; in particular it means that r knows the terms stored in the lists ak (*acquired knowledge*) and ik (*initial knowledge*) at step j of session c, and—if $j \neq 0$—also that a message from s to r is awaited for step j to be executed.

Initial States. The initial state of the system is:[4]

$$w(0, a, a, [\,], [a, b, k_{ab}], \underline{1}) \centerdot w(1, a, b, [\,], [b, a, k_{ab}], \underline{1}) \tag{1}$$

$$\centerdot\, w(0, b, b, [\,], [b, a, k_{ab}], \underline{2}) \centerdot w(1, b, a, [\,], [a, b, k_{ab}], \underline{2}) \tag{2}$$

$$\centerdot\, \mathit{fresh}(nc(na, \underline{1})) \centerdot \mathit{fresh}(nc(na, s(\underline{1}))) \tag{3}$$

$$\centerdot\, \mathit{fresh}(nc(na, \underline{2})) \centerdot \mathit{fresh}(nc(na, s(\underline{2}))) \tag{4}$$

$$\centerdot\, i(a) \centerdot i(b) \tag{5}$$

Facts (1) represent the initial state of principals a and b (as initiator and responder, resp.) in session $\underline{1}$. Dually, facts (2) represent the initial state of principals b and a (as responder and initiator, resp.) in session $\underline{2}$. Facts (3) and (4) state the initial freshness of the nonces. Facts (5) represent the information initially known by the intruder.

Rewrite rules over sets of facts are used to specify the transition system evolves.

Protocol Rules. The following rewrite rule models the activity of sending the first message:

$$\begin{aligned} & w(0, A, A, [\,], [A, B, K_{ab}], C) \centerdot \mathit{fresh}(nc(na, C)) \\ & \xrightarrow{step_1(A,B,C,K_{ab})} m(1, A, B, \{nc(na, C)\}_{K_{ab}}) \\ & \qquad \centerdot\, w(2, B, A, [nc(na, C)], [A, B, K_{ab}], C) \end{aligned} \tag{6}$$

Notice that nonce $nc(na, C)$ is added to the acquired knowledge of A for subsequent use. The receipt of the message and the reply of the responder is modeled by:

$$\begin{aligned} & m(1, A, B, \{nc(ID, C1)\}_{K_{ab}}) \\ & \centerdot\, w(1, A, B, [\,], [B, A, K_{ab}], C) \xrightarrow{step_2(A,B,C,C1,K_{ab},ID)} \\ & \qquad m(2, B, A, \{f(nc(ID, C1))\}_{K_{ab}}) \\ & \qquad \centerdot\, w(1, A, B, [\,], [B, A, K_{ab}], s(C)) \end{aligned} \tag{7}$$

The final step of the protocol is modeled by:

$$\begin{aligned} & m(2, B, A, \{f(nc(ID, C1))\}_{K_{ab}}) \\ & \quad \centerdot\, w(2, B, A, [nc(ID, C1)], [A, B, K_{ab}], C) \\ & \qquad \xrightarrow{step_3(A,B,C,C1,K_{ab},ID)} \\ & \qquad \centerdot\, w(0, A, A, [\,], [A, B, K_{ab}], s(C)) \end{aligned} \tag{8}$$

[3] As we will see, since the intruder may fake other principals' identity, the message might have been sent by the intruder.

[4] To improve readability we use the "$\centerdot$" operator as set constructor. For instance, we write "$x \centerdot y \centerdot z$" to denote the set $\{x, y, z\}$.

Intruder Rules. There are also rules specifying the behavior of the intruder. In particular the intruder is based on the model of Dolev and Yao [11]. For instance, the following rule models the ability of the intruder of diverting the information exchanged by the honest participants:

$$m(J,S,R,T) \xrightarrow{divert(J,R,S,T)} i(S) \centerdot i(R) \centerdot i(T) \quad (9)$$

The ability of encrypting and decrypting messages is modeled by:

$$i(T) \centerdot i(K) \xrightarrow{encrypt(K,T)} i(T) \centerdot i(K) \centerdot i(\{T\}_K) \quad (10)$$

$$i(\{T\}_K) \centerdot i(K) \xrightarrow{decrypt(K,T)} i(K) \centerdot i(T) \quad (11)$$

Finally, the intruder can send arbitrary messages possibly faking somebody else's identity in doing so:

$$i(T) \centerdot i(S) \centerdot i(R) \xrightarrow{fake_1(R,S,T)} i(T) \centerdot i(S) \centerdot i(R) \centerdot m(1,S,R,T) \quad (12)$$

$$i(T) \centerdot i(S) \centerdot i(R) \xrightarrow{fake_2(R,S,T)} i(T) \centerdot i(S) \centerdot i(R) \centerdot m(2,S,R,T) \quad (13)$$

Bad States. A security protocol is intended to enjoy a a specific security property. In our example this property is the ability of authenticating Bob to Alice. A security property can be specified by providing a set of "bad" states, i.e. states whose reachability implies a violation of the property. For instance, it is easy to see that any state containing both $w(0,a,a,[\,],[a,b,k_{ab}],s(\underline{1}))$ (i.e. Alice has finished the first run of session $\underline{1}$) and $w(1,a,b,[\,],[b,a,k_{ab}],\underline{1})$ (i.e. Bob is still at the beginning of session $\underline{1}$) witnesses a violation of the expected authentication property of our simple protocol and therefore it should be considered as a bad state.

2.2 Protocol Insecurity Problems

The above concepts can be recast into the concept of protocol insecurity problem. A *protocol insecurity problem* is a tuple $\Xi = \langle \mathcal{S}, \mathcal{L}, \mathcal{R}, \mathcal{I}, \mathcal{B} \rangle$ where $\mathcal{S}$ is a set of atomic formulae of a sorted first-order language called *facts*, $\mathcal{L}$ is a set of function symbols called *rule labels*, and $\mathcal{R}$ is a set of rewrite rules of the form $L \xrightarrow{\ell} R$, where L and R are finite subsets of $\mathcal{S}$ such that the variables occurring in R occur also in L, and ℓ is an expression of the form $l(\vec{x})$ where $l \in \mathcal{L}$ and $\vec{x}$ is the vector of variables obtained by ordering lexicographically the variables occurring in L. Let S be a state and $(L \xrightarrow{\ell} R) \in \mathcal{R}$, if σ is a substitution such that $L\sigma \subseteq S$, then one possible next state of S is $S' = (S \setminus L\sigma) \cup R\sigma$ and we indicate this with $S \xrightarrow{\ell\sigma} S'$. We assume the rewrite rules are *deterministic* i.e. if $S \xrightarrow{\ell\sigma} S'$ and $S \xrightarrow{\ell\sigma} S''$, then $S' \equiv S''$. The components $\mathcal{I}$ and $\mathcal{B}$ of a protocol insecurity problem are the initial state and a sets of states whose elements represent the bad states of the protocol respectively. A *solution to a protocol insecurity problem* Ξ (i.e. an attack to the protocol) is a sequence of states $S_1, \ldots, S_n$ such that $S_i \xrightarrow{\ell_i\sigma_i} S_{i+1}$ for $i = 1, \ldots, n$ and $\mathcal{I} \equiv S_1$, and there exists $S_{\mathcal{B}} \in \mathcal{B}$ such that $S_{\mathcal{B}} \subseteq S_n$.

3 Automatic SAT-Compilation of Protocol Insecurity Problems

Our proposed reduction of protocol insecurity problems to propositional logic is carried out in two steps. Protocol insecurity problems are first translated into planning problems which are in turn encoded into propositional formulae.

A *planning problem* is a tuple $\Pi = \langle \mathcal{F}, \mathcal{A}, Ops, I, G \rangle$, where $\mathcal{F}$ and $\mathcal{A}$ are disjoint sets of variable-free atomic formulae of a sorted first-order language called *fluents* and *actions* respectively; Ops is a set of expressions of the form

$$op(Act, Pre, Add, Del)$$

where $Act \in \mathcal{A}$ and Pre, Add, and Del are finite sets of fluents such that $Add \cap Del = \emptyset$; I and G are boolean combinations of fluents representing the initial state and the final states respectively. A state is represented by a set of fluents. An action is applicable in a state S iff the action preconditions occur in S and the application of the action leads to a new state obtained from S by removing the fluents in Del and adding those in Add. A *solution to a planning problem* Π is a sequence of actions whose execution leads from the initial state to a final state and the precondition of each action appears in the state to which it applies.

3.1 Encoding Planning Problems into SAT

Let $\Pi = \langle \mathcal{F}, \mathcal{A}, Ops, I, G \rangle$ be a planning problem with finite $\mathcal{F}$ and $\mathcal{A}$ and let n be a positive integer, then it is possible to build a set of propositional formulae Φ^n_Π such that any model of Φ^n_Π corresponds to a partial-order plan of length n which can be linearized into a solution of Π. The encoding of a planning problem into a set of SAT formulae can be done in a variety of ways (see [17, 13] for a survey). The basic idea is to add an additional time-index to the actions and fluents to indicate the state at which the action begins or the fluent holds. Fluents are thus indexed by 0 through n and actions by 0 through $n-1$. If p is a fluent or an action and i is an index in the appropriate range, then $i\!:\!p$ is the corresponding time-indexed propositional variable.

The set of formulae Φ^n_Π is the smallest set (intended conjunctively) such that:

- **Initial State Axioms:** $0{:}I \in \Phi^n_\Pi$;
- **Goal State Axioms:** $n{:}G \in \Phi^n_\Pi$;
- **Universal Axioms:** for each $op(\alpha, \mathit{Pre}, \mathit{Add}, \mathit{Del}) \in Ops$ and $i = 0, \ldots, n-1$:

$$(i{:}\alpha \supset \bigwedge\{i{:}p \mid p \in \mathit{Pre}\}) \in \Phi^n_\Pi$$
$$(i{:}\alpha \supset \bigwedge\{(i+1){:}p \mid p \in \mathit{Add}\}) \in \Phi^n_\Pi$$
$$(i{:}\alpha \supset \bigwedge\{\neg(i+1){:}p \mid p \in \mathit{Del}\}) \in \Phi^n_\Pi$$

- **Explanatory Frame Axioms:** for all fluents f and $i = 0, \ldots, n-1$:

$$(i{:}f \wedge \neg(i+1){:}f) \supset \bigvee \{i{:}\alpha \mid op(\alpha, \mathit{Pre}, \mathit{Add}, \mathit{Del}) \in Ops, f \in \mathit{Del}\} \in \Phi^n_\Pi$$

$$(\neg i{:}f \wedge (i+1){:}f) \supset \bigvee \{i{:}\alpha \mid op(\alpha, \mathit{Pre}, \mathit{Add}, \mathit{Del}) \in Ops, f \in \mathit{Add}\} \in \Phi^n_\Pi$$

- **Conflict Exclusion Axioms:** for $i = 0, \ldots, n-1$:

$$\neg(i{:}\alpha_1 \wedge i{:}\alpha_2) \in \Phi^n_\Pi$$

for all $\alpha_1 \neq \alpha_2$ such that $op(\alpha_1, \mathit{Pre}_1, \mathit{Add}_1, \mathit{Del}_1) \in Ops$, $op(\alpha_2, \mathit{Pre}_2, \mathit{Add}_2, \mathit{Del}_2) \in Ops$, and $\mathit{Pre}_1 \cap \mathit{Del}_2 \neq \emptyset$ or $\mathit{Pre}_2 \cap \mathit{Del}_1 \neq \emptyset$.

It is immediate to see that the number of literals in Φ^n_Π is in $O(n|\mathcal{F}| + n|\mathcal{A}|)$. Moreover the number of Universal Axioms is in $O(nP_0|\mathcal{A}|)$ where P_0 is the maximal number of fluents mentioned in an operator (usually a small number); the number of Explanatory Frame Axioms is in $O(n|\mathcal{F}|)$; finally, the number of Conflict Exclusion Axioms is in $O(n|\mathcal{A}|^2)$.

3.2 Protocol Insecurity Problems as Planning Problems

Given a protocol insecurity problem $\Xi = \langle \mathcal{S}, \mathcal{L}, \mathcal{R}, \mathcal{I}, \mathcal{B} \rangle$, it is possible to build a planning problem $\Pi_\Xi = \langle \mathcal{F}_\Xi, \mathcal{A}_\Xi, Ops_\Xi, I_\Xi, G_\Xi \rangle$ such that each solution to Π_Ξ can be translated back to a solution to Ξ: $\mathcal{F}_\Xi$ is the set of facts $\mathcal{S}$; $\mathcal{A}_\Xi$ and Ops_Ξ are the smallest sets such that $\ell\sigma \in \mathcal{A}_\Xi$ and $op(\ell\sigma, L\sigma, R\sigma \setminus L\sigma, L\sigma \setminus R\sigma) \in Ops$ for all $(L \xrightarrow{\ell} R) \in \mathcal{R}$ and all ground substitutions σ; finally $I_\Xi = \bigwedge\{f \mid f \in \mathcal{I}\} \bigwedge\{\neg f \mid f \in \mathcal{S}, f \notin \mathcal{I}\}$ and $G_\Xi = \bigvee_{S_\mathcal{B} \in \mathcal{B}} \bigwedge\{f \mid f \in S_\mathcal{B}\}$. For instance, the actions associated to (6) are of the form:

$$\begin{array}{l} op(step_1(A, B, C, K_{ab}), \\ \quad [w(0, A, A, [\,], [A, B, K_{ab}], C), \\ \quad\ \mathit{fresh}(nc(na, C))], \\ \quad [m(1, A, B, \{nc(C)\}_{K_{ab}}), \\ \quad\ w(2, B, A, [nc(na, C)], [A, B, K_{ab}], C)], \\ \quad [w(0, A, A, [\,], [A, B, K_{ab}], C), \\ \quad\ \mathit{fresh}(nc(na, C))]) \end{array}$$

The reduction of protocol insecurity problems to planning problems paves the way to an automatic SAT-compilation of protocol insecurity problems. However a direct application of the approach (namely the reduction of a protocol insecurity problem Ξ to a planning problem Π_Ξ followed by a SAT-compilation of Π_Ξ) is not immediately applicable. We therefore devised a set of optimizations and constraints whose combined effects often succeed in drastically reducing the size of the SAT instances.

3.3 Optimizations

Language specialization. We recall that for the reduction to propositional logic described in Section 3.1 to be applicable the set of fluents and actions must be finite. Unfortunately, the protocol insecurity problems introduced in Section 2 have an infinite number of facts and rule instances, and therefore the corresponding planning problems have an infinite number of fluents and actions. However the language can be restricted to a finite one since the set of states reachable in n steps is obviously finite (as long as the initial states comprise a finite number of facts). To determine a finite language capable to express the reachable states, it suffices to carry out a static analysis of the protocol insecurity problem.

To illustrate, let us consider again the simple protocol insecurity problem presented above and let $n = 7$, then $\|\mathtt{int}\| = \{0, 1, 2\}$, $\|\mathtt{user}\| = \{a, b\}$, $\|\mathtt{iuser}\| = \{a, b, \mathtt{intruder}\}$, $\|\mathtt{key}\| = \{kab\}$, $\|\mathtt{nonceid}\| = \{na\}$, $\|\mathtt{session}\| = \bigcup_{i=0}^{\lfloor n/(k+1) \rfloor} s^i(\underline{1}) \bigcup_{i=0}^{\lfloor n/(k+1) \rfloor} s^i(\underline{2})$, where k is the number of protocol steps in a session run (in this case $k = 2$),[5] $\|\mathtt{number}\| = nc(\mathtt{nonceid}, \mathtt{session})$, $\|\mathtt{func}\| = \bigcup_{i=0}^{n-1} f^i(\mathtt{number})$, $\|\mathtt{text}\| = \|\mathtt{iuser}\| \cup \|\mathtt{key}\| \cup \|\mathtt{number}\| \cup \|\mathtt{func}\| \cup \{\mathtt{func}\}\mathtt{key}$.[6]

Moreover, we can safely replace `list_of text` with $[\mathtt{text}, \mathtt{text}, \mathtt{text}]$. The set of facts is then equal to $i(\mathtt{text}) \cup \mathit{fresh}(\mathtt{number}) \cup m(\mathtt{int}, \mathtt{iuser}, \mathtt{iuser}, \mathtt{text}) \cup w(\mathtt{int}, \mathtt{iuser}, \mathtt{user}, \mathtt{list\_of\ text}, \mathtt{list\_of\ text}, \mathtt{session})$ which consists of 10^{12} facts. This language is finite, but definitely too big for the practical applicability of the SAT encoding.

A closer look to the protocol reveals that the above language still contains many spurious facts. In particular the $m(\ldots)$, $w(\ldots)$, and $i(\cdot)$ can be specialized (e.g. by using specialized sorts to restrict the message terms to those messages which are allowed by the protocol). By analyzing carefully the facts of the form $m(\ldots)$ and $w(\ldots)$ occurring in the protocol rules of our exam-

[5] The bound on the number of steps implies a bound on the maximum number of possible session repetitions.

[6] If $S_1, \ldots, S_m$ and S' are sorts and f is a function symbol of arity $S_1, \ldots, S_m \rightarrow S'$, then $\|S_i\|$ is the set of terms of sort S_i and $f(S_1, \ldots, S_m)$ denotes $\{f(t_1, \ldots, t_m) \mid t_i \in \|S_i\|, i = 1, \ldots, m\}$.

ple we can restrict the sort `func` in such a way that $\|\texttt{func}\| = \{f(\texttt{number})\}$ and replace `list_of text` with $[\texttt{iuser}, \texttt{iuser}, \texttt{key}] \cup [\texttt{number}]$. Thanks to this optimization, the number of facts drops to $12,620$.

An other important language optimization borrowed from [15] splits message terms containing pairs of messages such as $m(j,s,r,\langle msg_1, msg_2\rangle)$ (where $\langle \_,\_\rangle$ is the pairing operator) into two message terms $m(j,s,r,msg_1,1)$ and $m(j,s,r,msg_2,2)$. (Due to the simplicity of the shown protocol, splitting messages has no impact on its language size.)

Fluent splitting. The second family of optimizations is based on the observation that in $w(j,s,r,ak,ik,c)$, the union of the first three arguments with the sixth form a key (in the data base theory sense) for the relation. This allows us to modify the language by replacing $w(j,s,r,ak,ik,c)$ with the conjunction of two new predicates, namely $wk(j,s,r,ak,c)$ and $inknw(j,s,r,ik,c)$. Similar considerations (based on the observation that the initial knowledge of a principal r does not depend on the protocol step j nor on principal s) allow us to simplify $inknw(j,s,r,ik,c)$ to $inknw(r,ik,c)$. Another effective improvement stems from the observation that ak and ik are lists. By using the set of new facts $wk(j,s,r,ak_1,1,c),\ldots,wk(j,s,r,ak_l,l,c)$ in place of $wk(j,s,r,[ak_1,\ldots,ak_l],c)$ the number of wk terms drops from $O(|\texttt{text}|^l)$ to $O(l|\texttt{text}|)$.[7] In the usual simple example the application of fluent splitting reduces the number of facts to $1,988$.

Exploiting static fluents. The previous optimization enables a new one. Since the initial knowledge of the honest principal does not change as the protocol execution makes progress, facts of the form $inknw(r,ik,c)$ occurring in the initial state are preserved in all the reachable states and those not occurring in the initial state will not be introduced. In the corresponding planning problem, this means that all the atoms $i : inknw(r,ik,c)$ can be replaced by $inknw(r,ik,c)$ for $i = 0,\ldots,n-1$ thereby reducing the number of propositional letters in the encoding. Moreover, since the initial state is unique, this transformation enables an off-line partial instantiation of the actions and therefore a simplification of the propositional formula.

Reducing the number of Conflict Exclusion Axioms. A critical issue in the propositional encoding technique described in Section 3.1 is the quadratic growth of the number of Conflict Exclusion Axioms in the number of actions. This fact often confines the applicability of the method to problems with a small number of actions. A way to lessen this difficulty is to reduce the number of conflicting axioms by considering the intruder knowledge as *monotonic*. Let f be a fact, S and S' be states, then we say that f is monotonic iff for all S if $f \in S$ and $S \rightarrow S'$, then $f \in S'$. Since a monotonic fluent never appears in the delete list of some action, then it cannot be a cause of a conflict. The idea here is to transform the rules so to make the facts of the form $i(\cdot)$ monotonic. The transformation on the rules is very simple as it amounts to adding the monotonic facts occurring in the left hand side of the rule to its right hand side. A consequence is that a monotonic fact simplifies the Explanatory Frame Axioms relative to it. The nice effect of this transformation is that the number of Conflict Exclusion Axioms generated by the associated planning problems drops dramatically.

[7] If S is a sort, then $|S|$ is the cardinality of $\|S\|$.

Impersonate. The observation that most of the messages generated by the intruder by means of (12) and (13) are rejected by the receiver as non-expected or ill-formed suggests to restrict these rules so that the intruder sends only messages matching the patterns expected by the receiver. For each protocol rule of the form:

$$\ldots m(j,s,r,t) \centerdot w(j,s,r,ak,ik,c) \centerdot \ldots \xrightarrow{step_i(\ldots)} \ldots$$

we use a new rule of the form:

$$\begin{aligned}&\ldots \centerdot w(j,s,r,ak,ik,c) \centerdot i(s) \centerdot i(r) \centerdot i(t') \centerdot \ldots \\ &\xrightarrow{impersonate_i(\ldots)} \ldots \centerdot m(j,s,r,t') \centerdot w(j,s,r,ak,ik,c) \\ &\qquad\qquad \centerdot i(s) \centerdot i(r) \centerdot i(t') \centerdot \ldots\end{aligned}$$

This rule states that if agent r is waiting for a message t from s and the intruder knows a term t' matching t, then the intruder can impersonate s and send t'. This optimization (borrowed from [16]) often reduces the number of rule instances in a dramatic way. In our example, this optimization step allows us to trade all the 1152 instances of (12) and (13) with 120 new rules.

It is easy to see that this transformation is correct as it preserves the existing attacks and does not introduce new ones.

Step compression. A very effective optimization, called *step compression* has been proposed in [9]. It consists of the idea of merging intruder with protocol rules. In particular, an impersonate rule:

$$\begin{aligned}&w(i,x_1,x_2,x_3,x_4,x_5) \centerdot i(x_1) \centerdot i(x_2) \centerdot i(x_6) \\ &\qquad \xrightarrow{impersonate_i(\ldots)} m(i,x_1,x_2,x_6) \\ &\quad \centerdot w(i,x_1,x_2,x_3,x_4,x_5) \centerdot i(x_1) \centerdot i(x_2) \centerdot i(x_6)\end{aligned} \quad (14)$$

a generic protocol step rule:

$$\begin{aligned}&w(i,y_1,y_2,y_3,y_4,y_5) \centerdot m(i,y_1,y_2,y_6) \xrightarrow{step_i(\ldots)} \\ &\qquad w(j,y_1,y_2,y_7,y_4,y_5) \centerdot m(i+1,y_2,y_1,y_8)\end{aligned} \quad (15)$$

and a divert rule:

$$m(i+1,z_1,z_2,z_3) \xrightarrow{divert_{i+1}(\ldots)} i(z_1) \centerdot i(z_2) \centerdot i(z_3) \tag{16}$$

can be replaced by the following rule:

$$w(i,x_1,x_2,x_3,x_4,x_5)\sigma \centerdot i(x_1)\sigma \centerdot i(x_2)\sigma \centerdot i(x_6)\sigma \xrightarrow{step\_comp_i(\ldots)\sigma} w(j,y_1,y_2,y_7,y_4,y_5)\sigma \centerdot i(z_1) \centerdot \sigma \centerdot i(z_2)\sigma \centerdot i(z_3)\sigma$$

where $\sigma = \sigma_1 \circ \sigma_2$ with $\sigma_1 = mgu(\{w(i,x_1,x_2,x_3,x_4,x_5) = w(i,y_1,y_2,y_3,y_4,y_5), m(i,x_1,x_2,x_6) = m(i,y_1,y_2,y_6)\})$ and $\sigma_2 = mgu(\{m(i+1,y_2,y_1,y_8) = m(i+1,z_1,z_2,z_3)\})$.

The rationale of this optimization is that we can safely restrict our attention to computation paths where (14), (15), and (16) are executed in this sequence without any interleaved action in between.

By applying this optimization we reduce both the number of facts (note that the facts of the form $m(\ldots)$ are no longer needed) and the number of rules as well as the number of steps necessary to find the attacks. For instance, by using this optimization the partial-order plan corresponding to the attack to the Needham-Schroeder Public Key (NSPK) protocol [21] has length 7 whereas if this optimization is disabled the length is 10, the numbers of facts decreases from 820 to 505, and the number of rules from 604 to 313.

3.4 Bounds and Constraints

In some cases in order to get encodings of reasonable size, we must supplement the above attack-preserving optimizations with the following bounding techniques and constraints. Even if by applying them we may loose some attacks, in our experience (cf. Section 4) this rarely occurs in practice.

Bounding the number of session runs. Let n and k be the bounds in the number of operation applications and in the number of protocol steps characterizing a protocol session respectively. Then the maximum number of times a session can be repeated is $\lfloor n/(k+1) \rfloor$. Our experience indicates that attacks usually require a number of session repetitions that is less than $\lfloor n/(k+1) \rfloor$. As a matter of fact two session repetitions are sufficient to find attacks to all the protocols we have analyzed so far. By using this optimization we can reduce the cardinality of the sort `session` (in the case of the NSPK protocol, we reduce it by a factor 1.5) and therefore the number of facts that depend on it.

Multiplicity of fresh terms. The number of fresh terms needed to find an attack is usually less than the number of fresh terms available. As a consequence, a lot of fresh terms allowed by the language associated with the protocol are not used, and many facts depending on them are allowed, but also not used. Often, one single fresh term for each fresh term identifier is sufficient for finding the attack. For instance the simple example shown above has the only fresh term identifier na and to use the only nonce $nc(na,\underline{1})$ is enough to detect the attack. Therefore, the basic idea of this constraint is to restrict the number of fresh terms available, thereby reducing the size of the language. For example, application of this constraint to the analysis of the NSPK protocol protocol preserves the detection of the attack and reduces the numbers of facts and rules from 313 to 87 and from 604 to 54 respectively. Notice that for some protocols such as the Andrew protocol [7] the multiplicity of fresh terms is necessary to detect the attack.

Constraining the rule variables. This constraint is best illustrated by considering the Kao-Chow protocol (see e.g. [7]):

(1) $A \rightarrow S : A, B, N_a$
(2) $S \rightarrow B : \{A, B, N_a, K_{ab}\}K_{as}, \{A, B, N_a, K_{ab}\}K_{bs}$
(3) $B \rightarrow A : \{A, B, N_a, K_{ab}\}K_{as}, \{N_a\}K_{ab}, N_b$
(4) $A \rightarrow B : \{N_b\}K_{ab}$

During the step (2) S sends B a pair of messages of which only the second component is accessible to B. Since B does not know K_{ab}, then B cannot check that the occurrence of A in the first component is equal to that inside the second. As a matter of fact, we might have different terms at those positions. The constraint amounts to imposing that the occurrences of A (as well as of B, N_a, and K_{ab}) in the first and in the second part of the message must coincide. Thus, messages of the form $\{a, b, nc(na, s(\underline{1}))\}kas, \{a, b, nc(nb, s(\underline{1}))\}kbs$ would be ruled out by the constraint. The application of this constraint allows us to get a feasible encoding of the Kao-Chow protocols in reasonable time. For instance, with this constraint disabled the encoding of the Kao Chow Repeated Authentication 1 requires more than 1 hour, otherwise it requires 16.34 seconds.

4 Implementation and Computer Experiments

We have implemented the above ideas in SATMC, a SAT-based Model-Checker for security protocol analysis. Given a protocol insecurity problem Ξ, a bound on the length of partial-order plan n, and a set of parameters specifying which bounds and constraints must be enabled (cf.

Section 3.4), SATMC first applies the optimizing transformations previously described to Ξ and obtains a new protocol insecurity problem Ξ', then Ξ' is translated into a corresponding planning problem $\Pi_{\Xi'}$ which is in turn compiled into SAT using the methodology outlined in Section 3.1. The propositional formula is then fed to a state-of-the-art SAT solver (currently Chaff [20], SIM [14], and SATO [24] are supported) and any model found by the solver is translated back into an attack which is reported to the user.

SATMC is one of the back-ends of the AVISS tool [2]. Using this tool, the user can specify a protocol and the security properties to be checked using a high-level specification language and the tool translates the specification in an Intermediate Format (IF) based on multiset rewriting. The notion of protocol insecurity problem given in this paper is inspired by the Intermediate Format. Some of the features supported by the IF (e.g. public and private keys, compound keys as well as other security properties such as authentication and secrecy) have been neglected in this paper for the lack of space. However, they are supported by SATMC.

We have run our tool against a selection of problems drawn from [7]. The results of our experiments are reported in Table 1 and they are obtained by applying all the previously described optimizations, by setting $n = 10$, by imposing two session runs for session, by allowing multiple fresh terms, and by constraining the rule variables. For each protocol we give the kind of the attack found (Attack), the number of propositional variables (Atoms) and clauses (Clauses), and the time spent to generate the SAT formula (EncT) as well as the time spent by Chaff to solve the corresponding SAT instance (SolveT). The label MO indicates a failure to analyze the protocol due to memory-out.[8] It is important to point out that for the experiments we found it convenient to disable the generation of Conflict Exclusion Axioms during the generation of the propositional encoding. Of course, by doing this, we are no longer guaranteed that the solutions found are linearizable and hence executable. SATMC therefore checks any partial order plan found for executability. Whenever a conflict is detected, a set of clauses excluding these conflicts are added to the propositional formula and the resulting formula is fed back to the SAT-solver. This procedure is repeated until an executable plan is found or no other models are found by the SAT solver. This heuristics reduces the size of the propositional encoding (it does not create the Conflicts Exclusion Axioms) and it can also reduce the computation time whenever the time required to perform the executability checks is less than the time required for generating the Conflict Exclusion Axioms. The experiments show that the SAT solving activity is carried out very quickly and that the overall time is dominated by the SAT encoding.

[8] Times have been obtained on a PC with a 1.4 GHz Processor and 512 MB of RAM. Due to a limitation of SICStus Prolog the SAT-based model-checker is bound to use 128 MB during the encoding generation.

5 Conclusions and Perspectives

We have proposed an approach to the translation of protocol insecurity problems into propositional logic based on the combination of a reduction to planning and well-known SAT-reduction techniques developed for planning. Moreover, we have introduced a set of optimizing transformations whose application to the input protocol insecurity problem drastically reduces the size of the corresponding propositional encoding. We have presented SATMC, a model-checker based on our ideas, and shown that attacks to a set of well-known authentication protocols are quickly found by state-of-the-art SAT solvers.

Since the time spent by SAT solver is largely dominated by the time needed to generate the propositional encoding, in the future we plan to keep working on ways to reduce the latter. A promising approach amounts to treating properties of cryptographic operations as invariants. Currently these properties are modeled as rewrite rules (cf. rule (10) in Section 2.1) and this has a bad impact on the size of the final encoding. A more natural way to deal with these properties amounts to building them into the encoding but this requires, among other things, a modification of the explanatory frame axioms and hence more work (both theoretical and implementational) is needed to exploit this very promising transformation.

Moreover, we would like to experiment SATMC against security problems with partially defined initial states. Problems of this kind occur when the initial knowledge of the principal in not completely defined or when the session instances are partially defined. We conjecture that neither the size of the SAT encoding nor the time spent by the SAT solver to check the SAT instances will be significantly affected by this generalization. But this requires some changes in the current implementation of SATMC and a thorough experimental analysis.

References

[1] Luigia Carlucci Aiello and Fabio Massacci. Verifying security protocols as planning in logic programming. *ACM Transactions on Computational Logic*, 2(4):542–580, October 2001.

[2] A. Armando, D. Basin, M. Bouallagui, Y. Chevalier, L. Compagna, S. Moedersheim, M. Rusinowitch, M. Turuani, L. Viganò, and L. Vigneron. The AVISS Security Protocols Analysis Tool. In *14th International Conference on Computer-Aided Verification (CAV'02)*. 2002.

[3] David Basin and Grit Denker. Maude versus haskell: an experimental comparison in security protocol

Table 1: Performance of SATMC

Protocol	**Attack**	**Atoms**	**Clauses**	**EncT**	**SolveT**
ISO symmetric key 1-pass unilateral authentication	Replay	679	2,073	0.18	0.00
ISO symmetric key 2-pass mutual authentication	Replay	1,970	7,382	0.43	0.01
Andrew Secure RPC Protocol	Replay	161,615	2,506,889	80.57	2.65
ISO CCF 1-pass unilateral authentication	Replay	649	2,033	0.17	0.00
ISO CCF 2-pass mutual authentication	Replay	2,211	10,595	0.46	0.00
Needham-Schroeder Conventional Key	Replay STS	126,505	370,449	29.25	0.39
Woo-Lam Π	Parallel-session	7,988	56,744	3.31	0.04
Woo-Lam Mutual Authentication	Parallel-session	771,934	4,133,390	1,024.00	7.95
Needham-Schroeder Signature protocol	MM	17,867	59,911	3.77	0.05
Neuman Stubblebine repeated part	Replay STS	39,579	312,107	15.17	0.21
Kehne Langendorfer Schoenwalder (repeated part)	Parallel-session	-	-	MO	-
Kao Chow Repeated Authentication, 1	Replay STS	50,703	185,317	16.34	0.17
Kao Chow Repeated Authentication, 2	Replay STS	586,033	1,999,959	339.70	2.11
Kao Chow Repeated Authentication, 3	Replay STS	1,100,428	6,367,574	1,288.00	MO
ISO public key 1-pass unilateral authentication	Replay	1,161	3,835	0.32	0.00
ISO public key 2-pass mutual authentication	Replay	4,165	23,883	1.18	0.01
Needham-Schroeder Public Key	MM	9,318	47,474	1.77	0.05
Needham-Schroeder Public Key with key server	MM	11,339	67,056	4.29	0.04
SPLICE/AS Authentication Protocol	Replay	15,622	69,226	5.48	0.05
Encrypted Key Exchange	Parallel-session	121,868	1,500,317	75.39	1.78
Davis Swick Private Key Certificates, protocol 1	Replay	8,036	25,372	1.37	0.02
Davis Swick Private Key Certificates, protocol 2	Replay	12,123	47,149	2.68	0.03
Davis Swick Private Key Certificates, protocol 3	Replay	10,606	27,680	1.50	0.02
Davis Swick Private Key Certificates, protocol 4	Replay	27,757	96,482	8.18	0.13

Legenda: MM: Man-in-the-middle attack Replay STS: Replay attack based on a Short-Term Secret
MO: Memory Out

analysis. In Kokichi Futatsugi, editor, *Electronic Notes in Theoretical Computer Science*, volume 36. Elsevier Science Publishers, 2001.

[4] D. Bolignano. Towards the formal verification of electronic commerce protocols. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 133–146. 1997.

[5] Common Authentication Protocol Specification Language. URL `http://www.csl.sri.com/~millen/capsl/`.

[6] Cervesato, Durgin, Mitchell, Lincoln, and Scedrov. Relating strands and multiset rewriting for security protocol analysis. In *PCSFW: Proceedings of The 13th Computer Security Foundations Workshop*. IEEE Computer Society Press, 2000.

[7] John Clark and Jeremy Jacob. A Survey of Authentication Protocol Literature: Version 1.0, 17. Nov. 1997. URL `http://www.cs.york.ac.uk/~jac/papers/drareview.ps.gz`.

[8] Ernie Cohen. TAPS: A first-order verifier for cryptographic protocols. In *Proceedings of The 13th Computer Security Foundations Workshop*. IEEE Computer Society Press, 2000.

[9] Sebastian Moedersheim David Basin and Luca Viganò. An on-the-fly model-checker for security protocol analysis. forthcoming, 2002.

[10] Grit Denker, Jonathan Millen, and Harald Rueß. The CAPSL Integrated Protocol Environment. Technical Report SRI-CSL-2000-02, SRI International, Menlo Park, CA, October 2000. Available at `http://www.csl.sri.com/~millen/capsl/`.

[11] Danny Dolev and Andrew Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29), 1983.

[12] B. Donovan, P. Norris, and G. Lowe. Analyzing a library of security protocols using Casper and FDR. In *Proceedings of the Workshop on Formal Methods and Security Protocols*. 1999.

[13] Michael D. Ernst, Todd D. Millstein, and Daniel S. Weld. Automatic SAT-compilation of planning problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1169–1177. Morgan Kaufmann Publishers, San Francisco, 1997.

[14] Enrico Giunchiglia, Marco Maratea, Armando Tacchella, and Davide Zambonin. Evaluating search heuristics and optimization techniques in propositional satisfiability. In Rajeev Goré, Aleander Leitsch, and Tobias Nipkow, editors, *Proceedings of IJCAR'2001*, LNAI 2083, pages 347–363. Springer-Verlag, Heidelberg, 2001.

[15] Mei Lin Hui and Gavin Lowe. Fault-preserving simplifying transformations for security protocols. *Journal of Computer Security*, 9(1/2):3–46, 2001.

[16] Florent Jacquemard, Michael Rusinowitch, and Laurent Vigneron. Compiling and Verifying Security Protocols. In M. Parigot and A. Voronkov, editors, *Proceedings of LPAR 2000*, LNCS 1955, pages 131–160. Springer-Verlag, Heidelberg, 2000.

[17] Henry Kautz, David McAllester, and Bart Selman. Encoding plans in propositional logic. In Luigia Carlucci Aiello, Jon Doyle, and Stuart Shapiro, editors, *KR'96: Principles of Knowledge Representation and Reasoning*, pages 374–384. Morgan Kaufmann, San Francisco, California, 1996.

[18] Gawin Lowe. Casper: a compiler for the analysis of security protocols. *Journal of Computer Security*, 6(1):53–84, 1998. See also `http://www.mcs.le.ac.uk/~gl7/Security/Casper/`.

[19] Catherine Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996. See also `http://chacs.nrl.navy.mil/projects/crypto.html`.

[20] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*. 2001.

[21] R. M. (Roger Michael) Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. Technical Report CSL-78-4, Xerox Palo Alto Research Center, Palo Alto, CA, USA, 1978. Reprinted June 1982.

[22] L.C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1):85–128, 1998.

[23] D. Song. Athena: A new efficient automatic checker for security protocol analysis. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop (CSFW '99)*, pages 192–202. IEEE Computer Society Press, 1999.

[24] H. Zhang. SATO: An efficient propositional prover. In William McCune, editor, *Proceedings of CADE 14*, LNAI 1249, pages 272–275. Springer-Verlag, Heidelberg, 1997.

Session:
Security Protocols

Identifying Potential Type Confusion in Authenticated Messages

Catherine Meadows
Code 5543
Naval Research Laboratory
Washington, DC 20375
meadows@itd.nrl.navy.mil

Abstract

A *type confusion attack* is one in which a principal accepts data of one type as data of another. Although it has been shown by Heather et al. that there are simple formatting conventions that will guarantee that protocols are free from simple type confusions in which fields of one type are substituted for fields of another, it is not clear how well they defend against more complex attacks, or against attacks arising from interaction with protocols that are formatted according to different conventions. In this paper we show how type confusion attacks can arise in realistic situations even when the types are explicitly defined in at least some of the messages, using examples from our recent analysis of the Group Domain of Interpretation Protocol. We then develop a formal model of types that can capture potential ambiguity of type notation, and outline a procedure for determining whether or not the types of two messages can be confused. We also discuss some open issues.

1 Introduction

Type confusion attacks arise when it is possible to confuse a message containing data of one type with a message containing data of another. The most simple type confusion attacks are ones in which fields of one type are confused with fields of another type, such as is described in [7], but it is also possible to imagine attacks in which fields of one type are confused with a concatenation of fields of another type, as is described by Snekkenes in [8], or even attacks in which pieces of fields of one type are confused with pieces of fields of other types.

Simple type confusion attacks, in which a field of one type is confused with a field of another type, are easy to prevent by including type labels (tags) for all data and authenticating labels as well as data. This has been shown by Heather et al. [4], in which it is proved that, assuming a Dolev-Yao-type model of a cryptographic protocol and intruder, it is possible to prevent such simple type confusion attacks by the use of this technique. However, it is not been shown that this technique will work for more complex type confusion attacks, in which tags may be confused with data, and terms or pieces of terms of one type may be confused with concatenations of terms of several other types.[1] More importantly, though, although a tagging technique may work within a single protocol in which the technique is followed for all authenticated messages, it does not prevent type confusion of a protocol that uses the technique with a protocol that does not use the technique, but that does use the same authentication keys. Since it is not uncommon for master keys (especially public keys) to be used with more than one protocol, it may be necessary to develop other means for determining whether or not type confusion is possible. In this paper we explore these issues further, and describe a procedure for detecting the possibility of the more complex varieties of type confusion.

The remainder of this paper is organized as follows. In order to motivate our work, in Section Two, we give a brief account of a complex type confusion flaw that was recently found during an analysis of the Group Domain of Authentication Protocol, a secure multicast protocol being developed by the Internet Engineering Task Force. In Section Three we give a formal model for the use of types in protocols that takes into account possible type ambiguity. In Section Four we describe various techniques for constructing the artifacts that will be used in our procedure. In Section Five we give a procedure for determining whether it is possible to confuse the type of two messages. In Section Six we illustrate our procedure by showing how it could be applied to a simplified version of GDOI. In Section Seven we conclude the paper and give suggestions for further research.

2 The GDOI Attack

In this section we describe a type flaw attack that was found on an early version of the GDOI protocol.

The Group Domain of Interpretation protocol (GDOI) [2], is a group key distribution protocol that is undergoing the IETF standardization process. It is built on top

[1] We believe that it could, however, if the type tags were augmented with tags giving the length of the tagged field, as is done in many implementations of cryptographic protocols.

of the ISAKMP [6] and IKE [3] protocols for key management, which imposes some constraints on the way in which it is formatted. GDOI consists of two parts. In the first part, called the Groupkey Pull Protocol, a principal joins the group and gets a group key-encryption-key from the Group Controller/Key Distributor (GCKS) in a handshake protocol protected by a pairwise key that was originally exchanged using IKE. In the second part, called the Groupkey Push Message, the GCKS sends out new traffic encryption keys protected by the GCKS's digital signature and the key encryption key.

Both pieces of the protocol can make use of digital signatures. The Groupkey Pull Protocol offers the option of including a Proof-of-Possession field, in which either or both parties can prove possession of a public key by signing the concatenation of a nonce NA generated by the group member and a nonce NB generated by the GCKS. This can be used to show linkage with a certificate containing the public key, and hence the possession of any identity or privileges stored in that certificate.

As for the Groupkey Push Message, it is first signed by the GCKS's private key, and then encrypted with the key encryption key. The signed information includes a header HDR, (which is sent in the clear), and contains, besides the header, the following information:

1. a sequence number SEQ (to guard against replay attacks);
2. a security association SA;
3. a Key Download payload KD, and;
4. an optional certificate, CERT.

According to the conventions of ISAKMP, HDR must begin with a random or pseudo-random number. In pairwise protocols, this is jointly generated by both parties, but in GDOI, since the message must go from one to many, this is not practical. Thus, the number is generated by the GCKS. Similarly, it is likely that the Key Download message will end in a random number: a key. Thus, it is reasonable to assume that the signed part of a Groupkey Push Message both begins and ends in a random number.

We found two type confusion attacks. In both, we assume that the same private key is used by the GCKS to sign POPs and Groupkey Push Messages. In the first of these, we assume a dishonest group member who wants to pass off a signed POP from the GCKS as a Groupkey Push Message. To do this, we assume that she creates a fake plaintext Groupkey Push Message GPM, which is missing only the last (random) part of the Key Download Payload. She then initiates an instance of the Groupkey Pull Protocol with the GCKS, but in place of her nonce, she sends GPM. The GCKS responds by appending its nonce NB and signing it, to create a signed (GPM,NB). If NB is of the right size, this will look like a signed Groupkey Push Message. The group member can then encrypt it with the key encryption key (which she will know, being a group member) and send it out to the entire group.

The second attack requires a few more assumptions. We assume that there is a group member A who can also act as a GCKS, and that the pairwise key between A and another GCKS, B, is stolen, but that B's private key is still secure. Suppose that A, acting as a group member, initiates a Groupkey Pull Protocol with B. Since their pairwise key is stolen, it is possible for an intruder I to insert a fake nonce for B's nonce NB. The nonce he inserts is a fake Groupkey Push Message GPM' that it is complete except for a prefix of the header consisting of all or part of the random number beginning the header. A then signs (NA,GPM'), which, if NA is of the right length, will look like the signed part of a Groupkey Push Message. The intruder can then find out the key encryption key from the completed Groupkey Pull Protocol and use it to encrypt the resulting (NA,GPM') to create a convincing fake Groupkey Push Message.

Fortunately, the fix was simple. Although GDOI was constrained by the formatting required by ISAKMP, this was not the case for the information that was signed within GDOI. Thus, the protocol was modified so that, whenever a message was signed within GDOI, information was prepended saying what the purpose was (e.g. a member's POP, or a Groupkey Push Message). This eliminated the type confusion attacks.

There are several things to note here. The first is that existing protocol analysis tools are not very good at finding these types of attacks. Most assume that some sort of strong typing is already implemented. Even when this is not the case, the ability to handle the various combinations that arise is somewhat limited. For example, we found the second, less feasible, attack automatically with the NRL Protocol Analyzer, but the tool could not have found the first attack, since the ability to model it requires the ability to model the associativity of concatenation, which the NRL Protocol Analyzer lacks. Moreover, type confusion attacks do not require a perfect matching between fields of different types. For example, in order for the second attack to succeed, it is not necessary for NA to be the same size as the random number beginning the header, only that it be no longer than that number. Again, this is something that is not within the capacity of most crypto protocol analysis tools. Finally, most crypto protocol analysis tools are not equipped for probabilistic analysis, so they would not be able to find cases in which, although type confusion would not be possible every time, it would occur with a high enough probability to be a concern.

The other thing to note is that, as we said before, even though it is possible to construct techniques that can be used to guarantee that protocols will not interact insecurely with other protocols that are formatted using the same technique, it does not mean that they will not interact insecurely with protocols that were formatted using differ-

ent techniques, especially if, in the case of GDOI's use of ISAKMP, the protocol wound up being used differently than it was originally intended (for one-to-many instead of pairwise communication). Indeed, this is the result one would expect given previous results on protocol interaction [5, 1]. Since it is to be expected that different protocols will often use the same keys, it seems prudent to investigate to what extent an authenticated message from one protocol could be confused with an authenticated message from another, and to what extent this could be exploited by a hostile intruder. The rest of this paper will be devoted to the discussion of a procedure for doing so.

3 The Model

In this section we will describe the model that underlies our procedure. It is motivated by the fact that different principals may have different capacities for checking types of messages and fields in messages. Some information, like the length of the field, may be checkable by anybody. Other information, like whether or not a field is a random number generated by a principal, or a secret key belonging to a principal, will only be checkable by the principal who generated the random number in the first case, and by the possessor(s) of the secret key in the second place. In order to do this, we need to develop a theory of types that take differing capacities for checking types into account.

We assume an environment consisting of principals who possess information and can check properties of data based on that information. Some information is public and is shared by all principals. Other information may belong to only one or a few principals.

Definition 3.1 *A* field *is a sequence of bits. We let ι denote the empty field. If x and y are two fields, we let $x||y$ denote the concatenation of x and y. If $\bar{x}$ and $\bar{y}$ are two lists of fields, then we let $append(\bar{x}, \bar{y})$ denote the list obtained by appending $\bar{y}$ to $\bar{x}$.*

Definition 3.2 *A* type *is a set of fields, which may or may not have a probability distribution attached. If P is a principal, then a* type local to P *is a type such that membership in that type is checkable by P. A* public type *is one whose membership is checkable by all principals. If G is a group of principals, then a* type private to G *is a type such that membership in that type is checkable by the members of G and only the members of G.*

Examples of a public type would be all strings of length 256, the string "key," or well-formed IP addresses. Examples of private types would be a random nonce generated by a principal (private to that principal) a principal's private signature key (private to that principal), and a secret key shared by Alice and Bob (private to Alice and Bob, and perhaps the server that generated the key, if one exists). Note that a private type is not necessarily secret; all that is required is that only members of the group to whom the type is private have a guaranteed means of checking whether or not a field belongs to that type. As in the case of the random number generated by a principal, other principals may have been told that a field belongs to the type, but they do not have a reliable means of verifying this.

The decision as to whether or not a type is private or public may also depend upon the protocol in which it is used and the properties that are being proved about the protocol. For example, to verify the security of a protocol that uses public keys to distribute master keys, we may want to assume that a principal's public key is a public type, while if the purpose of the protocol is to validate a principal's public key, we may want to assume that the type is private to that principal and some certification authority. If the purpose of the protocol is to distribute the public key to the principal, we may want to assume that the type is private to the certification authority alone.

Our use of public and local types is motivated as follows. Suppose that an intruder wants to fool Bob into accepting an authenticated message M from a principal Alice as an authenticated message N from Alice. Since M is generated by Alice, it will consist of types local to her. Thus, for example, if M is supposed to contain a field generated by Alice it will be a field generated by her, but if it is supposed to contain a field generated by another party, Alice may only be able to check the publically available information such as the formatting of that field before deciding to include it in the message. Likewise, if Bob is verifying a message purporting to be N, he will only be able to check for the types local to himself. Thus, our goal is to be able to check whether or not a message built from types local to Alice can be confused with another message built from types local to Bob, and from there, to determine if an intruder is able to take advantage of this to fool Bob into producing a message that can masquerade as one from Alice.

We do not attempt to give a complete model of an intruder in this paper, but we do need to have at at least some idea of what types mean from the point of view of the intruder to help us in computing the probability of an intruder's producing type confusion attacks. In particular, we want to determine the probability that the intruder can produce (or force the protocol to produce) a field of one type that also belongs to another type. Essentially, there are two questions of interest to an intruder: given a type, can it control what field of that type is sent in a message, and given a type, will any arbitrary member of that type be accepted by a principal, or will a member be accepted only with a certain probability.

Definition 3.3 *We say that a type is* under the control of the intruder *if there is no probability distribution associated with it. We say that a type is* probabilistic *if there a a probability distribution associated with it. We say that a probabilistic type local to a principal A is* under the con-

trol of A *if the probability of* A *accepting a field as a member of* X *is given by the probability distribution associated with* X.

The idea behind probabilistic types and types under control of the intruder is that the intruder can choose what member of a type can be used in a message if it is under its control, but for probabilistic types the field used will be chosen according to the probability distribution associated with the type. On the other hand, if a type is not under the control of a principal A, then A will accept any member of that type, while if the type is under the control of A, she will only accept an element as being a member of that type according to the probability associated with that type.

An example of a type under the control of an intruder would be a nonce generated by the intruder, perhaps while impersonating someone else. An example of a probabilistic type that is not under the control of A would be a nonce generated by another principal B and sent to A in a message. An example of a probabilistic type that is also under the control of A would be a nonce generated by A and sent by A in a message, or received by A in some later message.

Definition 3.4 *Let* X *and* Y *be two types. We say that* $X \sqcap Y$ *holds if an intruder can force a protocol to produce an element* x *of* X *that is also an element of* Y.

Of course, we are actually interested in the probability that $X \sqcap Y$ holds. Although the means for calculating $P(X \sqcap Y)$ may vary, we note that the following holds if there are no other constraints on X and Y:

1. If X and Y are both under the control of the intruder, then $P(X \sqcap Y)$ is 1 if $X \cap Y \neq \phi$ and is zero otherwise;
2. If X is under the control of the intruder, and Y is a type under the control of A, and the intruder knows the value of the member of Y before choosing the member of X, then $P(Y \sqcap X) = P(\hat{x} \in X \cap Y)$, where $\hat{x}$ is the random variable associated with X;
3. If X a type under the control of A, and Y is a type local to B but not under the control of B, then $P(X \sqcap Y) = P(\hat{x} \in X \cap Y)$;
4. If X is under the control of A and Y is under the control of some other (non-intruder) B, then $P(Y \sqcap X) = P(\hat{x} = \hat{y})$ where $\hat{x}$ is the random variable associated with X, and $\hat{y}$ is the random variable associated with Y.

Now that we have a notion of type for fields, we extend it to a notion of type for messages.

Definition 3.5 *A* message *is a concatenation of one or more fields.*

Definition 3.6 *A* message type *is a function* $\mathcal{R}$ *from lists of fields to types, such that:*

1. *The empty list is in* $\mathrm{Dom}(\mathcal{R})$*;*
2. $\langle x_1, ..., x_k \rangle \in \mathrm{Dom}(\mathcal{R})$ *if and only if* $\langle x_1, ..., x_{k-1} \rangle \in \mathrm{Dom}(\mathcal{R})$ *and* $x_k \in \mathcal{R}(\langle x_1, ..., x_{k-1} \rangle)$*;*
3. *If* $\langle x_1, ..., x_k \rangle \in \mathrm{Dom}(\mathcal{R})$*, and* $x_k = \iota$*, then* $\mathcal{R}(\langle x_1, ..., x_k \rangle) = \{\iota\}$*, and ;*
4. *For any infinite sequence* $S = \langle ..., x_i, ... \rangle$ *such that all prefixes of* S *are in* $Dom(\mathcal{R})$*, there exists an* n *such that, for all* $i > n$*,* $x_i = \iota$*.*

The second part of the definition shows how, once the first $k-1$ fields of a message are known, then $\mathcal{R}$ can be used to predict the type of the k'th field. The third and fourth parts describe the use of the empty list ι in indicating message termination. The third part says that, if the message terminates, then it can't start up again. The fourth part says that all messages must be finite. Note, however, that it does not require that messages be of bounded length. Thus, for example, it would be possible to specify, say, a message type that consists of an unbounded list of keys.

The idea behind this definition is that the type of the n'th field of a message may depend on information that has gone before, but exactly where this information goes may depend upon the exact encoding system used. For example, in the tagging system in [4], the type is given by a tag that precedes the field. In many implementations, the tag will consist of two terms, one giving the general type (e.g. "nonce"), and the other giving the length of the field. Other implementations may use this same two-part tag, but it may not appear right before the field; for example in ISAKMP, and hence in GDOI, the tag refers, not to the field immediately following it, but the field immediately after that. However, no matter how tagging is implemented, we believe that it is safe to assume that any information about the type of a field will come somewhere before the field, since otherwise it might require knowledge about the field that only the tag can supply (such as where the field ends) in order to find the tag.

Definition 3.7 *The* support *of a message type* $\mathcal{R}$ *is the set of all messages of the form* $x_1||...||x_n$ *such that* $\langle x_1, ..., x_n \rangle \in \mathrm{Dom}(\mathcal{R})$.

For an example of a message type, we consider a message of the form

$$\text{“}nonce\text{''}, N_1, NONCE_1, \text{“}nonce\text{''}, N_2, NONCE_2$$

where $NONCE_1$ is a random number of length N_1 generated by the creator of the message, N_1 is a 16-bit integer, and $NONCE_2$ is a random number of length N_2, where both $NONCE_2$ and N_2 are generated by the intended receiver, and N_2 is another 16-bit integer. ¿From the point of view of the generator of the message, the message type is as follows:

1. $\mathcal{R}(\langle\rangle) =$ "$nonce''$.
2. $\mathcal{R}(\langle$"$nonce''\rangle) = \{X|length(X) = 16\}$. Since N_1 is generated by the sender, it is a type under the control of the sender consisting of the set of 16-bit integers, with a certain probability attached.
3. $\mathcal{R}(\langle$"$nonce'', N_1\rangle) = \{X|length(X) = N_1\}$. Again, this is a private type consisting of the set of fields of length N_1. In this case, we can choose the probability distribution to be the uniform one.
4. $\mathcal{R}(\langle$"$nonce'', N_1, NONCE_1\rangle) = \{$"$nonce''\}$.
5. $\mathcal{R}(\langle$"$nonce'', N_1, NONCE_1,$ "$nonce''\rangle) = \{X|length(X) = 16\}$. Since the sender did not actually generate N_2, all he can do is check that it is of the proper length, 16. Thus, this type is not under the control of the sender. If N_2 was not authenticated, then it is under the control of the intruder.
6. $\mathcal{R}(\langle$"$nonce'', N_1, NONCE_1,$ "$nonce'', N_2\rangle) = \{Y|length(Y) = N_2\}$. Again, this value is not under the control of the sender, all the principal can do is check that what purports to be a nonce is indeed of the appropriate length.
7. $\mathcal{R}(\langle$"$nonce'', N_1, NONCE_1,$ "$nonce'', N_2,$ $NONCE_1, \rangle) = \{\iota\}$. This last tells us that the message ends here.

¿From the point of view of the receiver of the message, the message type will be somewhat different. The last two fields, N_2 and $NONCE_2$ will be types under the control of the receiver, while N_1 and $NONCE_1$ will be types not under its control, and perhaps under the control of the intruder, whose only checkable property is their length. This motivates the following definition:

Definition 3.8 *A* message type local to a principal P *is a message type $\mathcal{R}$ whose range is made up of types local to P.*

We are now in a position to define type confusion.

Definition 3.9 *Let $\mathcal{R}$ and $\mathcal{S}$ be two message types. We say that a pair of sequences $\langle x_1, ..., x_n\rangle \in Dom(\mathcal{R})$ and $\langle y_1, ..., y_n\rangle \in Dom(\mathcal{S})$ is a* type confusion between $\mathcal{R}$ and $\mathcal{S}$ *if:*

1. $\iota \in \mathcal{R}(\langle x_1, ..., x_n\rangle)$*;*
2. $\iota \in \mathcal{S}(\langle y_1, ..., y_m\rangle)$*, and;*
3. $x_1||...||x_n = y_1||...||y_m$.

The first two conditions say that the sequences describe complete messages. That last conditions says that the messages, considered as bit-strings, are identical.

Definition 3.10 *Let $\mathcal{R}$ and $\mathcal{S}$ be two message types. We say that $\mathcal{R} \sqcap \mathcal{S}$ holds if an intruder is able to force a protocol to produce an $\bar{x}$ in $Dom(\mathcal{R})$ such that there exists $\bar{y}$ in $Dom(\mathcal{S})$ such that $(\bar{x}, \bar{y})$ is a type confusion..*

Again, what we are interested in is computing, or at least estimating, $P(\mathcal{R} \sqcap \mathcal{S})$. This will be done in Section 5.

4 Constructing and Rearranging Message Types

In order to perform our comparison procedure, we will need the ability to build up and tear down message types, and create new message types out of old. In this section we describe the various ways that we can do this.

We begin by defining functions that are restrictions of message types (in particular to prefixes and postfixes of tuples).

Definition 4.1 *An* n-postfix message type *is a function $\mathcal{R}$ from tuples of length n or greater to types such that:*

1. *For all $k > 0$, $\langle x_1, ..., x_{n+k}\rangle \in \mathrm{Dom}(\mathcal{R})$ if and only if $x_{n+k} \in \mathcal{R}(\langle x_1, ..., x_{n+k-1}\rangle)$;*
2. *If $\langle x_1, ..., x_{n+k}\rangle \in \mathrm{Dom}(\mathcal{R})$, and $x_{n+k} = \iota$, then $\mathcal{R}(\langle x_1, ..., x_{n+k+1}\rangle) = \{\iota\}$, and ;*
3. *For any infinite sequence $S = \langle ..., x_i, ...\rangle$ such that all prefixes of S of length n and greater are in $Dom(\mathcal{R})$, there exists an m such that, for all $i > m$, $x_i = \iota$.*

We note that the restriction of a message type $\mathcal{R}$ to sequences of length n or greater is an n-postfix message type, and that a message type is a 0-postfix message type.

Definition 4.2 *An n-prefix message type is a function $\mathcal{R}$ from tuples of length less than n to types such that:*

1. *$\mathcal{R}$ is defined over the empty list;*
2. *For all $k < n$, $\langle x_1, ..., x_k\rangle \in \mathrm{Dom}(\mathcal{R})$ if and only if $x_k \in \mathcal{R}(\langle x_1, ..., x_{k-1}\rangle)$, and;*
3. *If $k < n - 1$, and $\langle x_1, ..., x_k\rangle \in \mathrm{Dom}(\mathcal{R})$, and $x_k = \iota$, then $\mathcal{R}(\langle x_1, ..., x_{k+1}\rangle) = \{\iota\}$.*

We note that the restriction of a message type to sequences of length less than n is an n-prefix message type.

Definition 4.3 *We say that a message type or n-prefix message type $\mathcal{R}$ is* t-bounded *if $\mathcal{R}(x) = \iota$ for all tuples x of length t or greater.*

In particular, a message type that is both t-bounded and t-postfix will be a trivial message type.

Definition 4.4 *Let $\mathcal{R}$ be an n-postfix message type. Let X be a set of m-tuples in the pre-image of $\mathcal{R}$, where m $\geq$ n. Then $\mathcal{R}\lfloor X$ is defined to be the restriction of R to the set of all $\langle x_1, ..., x_m, ..., x_r\rangle$ in $Dom(\mathcal{R})$ such that $\langle x_1, ..., x_m\rangle \in X$.*

Definition 4.5 *Let $\mathcal{R}$ be an n-prefix message type. Let X be a set of n-1 tuples. Then $\mathcal{R}\lceil X$ is defined to be the restriction of $\mathcal{R}$ to the set of all tuples $\bar{x}$ such that $\bar{x} \in X$, or $\bar{x} = \langle x_1, ...x_i\rangle$ such that there exists $\langle y_{i+1}, ..., y_{n-1}\rangle$ such that $\langle x_1, ...x_i, y_{i+1}, ..., y_{n-1}\rangle \in X$.*

Definition 4.6 *Let $\mathcal{R}$ be an n-postfix message type. Then $Split(\mathcal{R})$ is the function whose domain is the set of all $\langle x_1, ..., x_n, y_1, y_2, x_{n+2}, ..., x_m\rangle$ of length n+1 or greater such that $\langle x_1, ..., x_n, y_1||y_2, x_{n+2}, ..., x_m\rangle \in Dom(\mathcal{R})$ and such that*

a. For the tuples of length $i > n$ +1, $Split(\mathcal{R})(\langle x_1, ..., x_n, y_1, y_2, x_{n+2}, ..., X_m\rangle) = \mathcal{R}(\langle x_1, ..., x_n, y_1||y_2, x_{n+2}, ..., x_m\rangle)$, and;

b. For tuples of length n +1 , $Split(\mathcal{R})(\langle y_1, ..., y_{n+1}\rangle) = \{z \mid \langle y_1, ..., y_{n+1}||z\rangle \in Dom(\mathcal{R})$.

Definition 4.7 *Let $\mathcal{R}$ be an n-prefix message type. Let F be a function from a set of n-tuples to types such that there is at least one tuple $\langle x_{i+1}..., x_n\rangle$ in the domain of F such that $\langle x_{i+1}..., x_{n-1}\rangle$ is in the domain of $\mathcal{R}$. Then $\mathcal{R}\sharp F$, the* extension of $\mathcal{R}$ by *F, is the function whose domain is*

a. For $i < n$, the set of all $\langle x_1...., x_i\rangle$ such that $\langle x_1...., x_i\rangle \in Dom(\mathcal{R})$, and such that there exists $\langle x_{i+1}..., x_n\rangle$ such that $\langle x_1...., x_i, x_{i+1}..., x_n\rangle \in Dom(F)$;

b For i = n, the set of all $\langle x_1...., x_{n-1}, x_n\rangle$ such that $\langle x_1...., x_{n-1}\rangle \in Dom(\mathcal{R})$ and $\langle x_1...., x_{n-1}, x_n\rangle \in Dom(F)$;

and whose restriction to tuples of length less than n is $\mathcal{R}$, and whose restriction to n-tuples is F.

Proposition 4.1 *If $\mathcal{R}$ is an n-postfix message type, then $\mathcal{R}\lfloor X$ is an m-postfix message type for any set of m-tuples X, and $Split(\mathcal{R})$ is an (n+1)-postfix message type. If $\mathcal{R}$ is t-bounded, then so is $\mathcal{R}\lfloor X$, while $Split(\mathcal{R})$ is (t+1)-bounded. Moreover, if S is an n-prefix message type, then so is $\mathcal{S}\lceil Y$ for any set of n-1 tuples Y, and $\mathcal{S}\sharp F$ is an (n+1)-prefix message type for any function F from n-tuples to types such at for at least one element $\langle x_{i+1}..., x_n\rangle$ in the domain of F, $\langle x_{i+1}..., x_{n-1}\rangle$ is in the domain of $\mathcal{S}$.*

We close with one final definition.

Definition 4.8 *Let F be a function from k-tuples of fields to types. We define $Pre(F)$ to be the function from k-tuples of fields to types defined by $Pre(F)(x)$ is the set of all prefixes of all elements of $F(x)$.*

5 The Zipper: A Procedure for Comparing Message Types

We now can define our procedure for determining whether or not type confusion is possible between two message types $\mathcal{R}$ and $\mathcal{S}$, that is, whether it is possible for a verifier to mistake a message of type $\mathcal{R}$ generated by some principal for a message of type $\mathcal{S}$ generated by that same principal , where $\mathcal{R}$ is a message type local to the generator, and $\mathcal{S}$ is a message type local to the verifier. But, in order for this to occur, the probability of $\mathcal{R} \sqcap \mathcal{S}$ must be nontrivial. For example, consider a case in which $\mathcal{R}$ is a type local to and under the control of Alice consisting of a random variable 64 bits long, and $\mathcal{S}$ consists of another random 64-bit variable local to and under the control of Bob. It is possible that $\mathcal{R} \sqcap \mathcal{S}$ holds, but the probability that this is so is only $1/2^{64}$. On the other hand, if $\mathcal{R}$ is under the control of the intruder, then the probability that their support is non-empty is one. Thus, we need to choose a threshold probability, such that we consider a type confusion whose probability falls below the threshold to be of negligible consequence.

Once we have chosen a threshold probability, our strategy will be to construct a "zipper"between the two message types to determine their common support. We will begin by finding the first type of $\mathcal{R}$ and the first type of $\mathcal{S}$, and look for their intersection. Once we have done this, for each element in the common support, we will look for the intersection of the next two possible types of $\mathcal{R}$ and $\mathcal{S}$, respectively, and so on. Our search will be complicated, however, by the fact that the matchup may not be between types, but between pieces of types. Thus, for example, elements of the first type of $\mathcal{R}$ may be identical to the prefixes of elements of the first type of $\mathcal{S}$, while the remainders of these elements may be identical to elements of the second type of $\mathcal{R}$, and so forth. So we will need to take into account three cases: the first, where two types have a nonempty intersection, the second, where a type from $\mathcal{R}$ (or a set of remainders of types from $\mathcal{R}$) has a nonempty intersection with a set of prefixes from the second type of $\mathcal{S}$, and the third, where a type from $\mathcal{S}$ (or a set of remainders of types from $\mathcal{S}$) has a nonempty intersection with a set of prefixes from the second type of $\mathcal{R}$. All of these will impose a constraint on the relative lengths of the elements of the types from $\mathcal{S}$ and $\mathcal{R}$, which need to be taken into account, since some conditions on lengths may be more likely to be satisfied than others.

Our plan is to construct our zipper by use of a tree in which each node has up to three possible child nodes, corresponding to the three possibilities given above. Let $\mathcal{R}$ and $\mathcal{S}$ be two message types, and let p be a number between 1 and 0, such that we are attempting to determine whether the probability of constructing a type confusion between $\mathcal{R}$ and $\mathcal{S}$ is greater than p. We define a tertiary tree of sept-tuples as follows. The first entry of each

sept-tuple is a set U of triples $\langle x, \bar{y}, \bar{z}\rangle$, where x is a bit-string and $\bar{y} = \langle y_1, ..., y_n\rangle$ and $\bar{z} = \langle z_1, ..., z_m\rangle$ such that $y1||...||y_n = z_1||...||z_m = x$. We will call U the *support* of the node. The second and third entries are n and m postfix message types, respectively. The fourth and fifth are message types or prefix message types. The sixth is a probability q. The seventh is a set of constraints on lengths of types. The root of the tree is of the form $\langle \phi, \mathcal{R}, \mathcal{S}, \langle\rangle, \langle\rangle, 1, D\rangle$, where D is the set of length constraints introduced by $\mathcal{R}$ and $\mathcal{S}$.

Given a node, $\langle U, \mathcal{H}, \mathcal{I}, \mathcal{J}, \mathcal{K}, q, C\rangle$, we construct up to three child nodes as follows:

1. The first node corresponds to the case in which a term from H can be confused with a term from I. Let T be the set of all $\langle x, \bar{y}, \bar{z}\rangle \in U$ such that $P(\mathcal{H}(\bar{y}) \cap \mathcal{I}(\bar{z}) \neq \phi) \cdot q > p$. Then, if T is non-empty, we construct a child node as follows:

   a. The first element of the new tuple is the set T' of all $\langle x', \bar{y}', \bar{z}'\rangle$ such that there exists $\langle x, \bar{y}, \bar{z}\rangle \in T$ such that $x' = x||y_1$, where $y_1 \in \mathcal{H}_n(\bar{y})$, $\bar{y}' = append(\bar{y}, \langle y_1\rangle)$, and $\bar{z}' = append(\bar{z}, \langle y_1\rangle)$;

   Note that, by definition y_1 is an element of $\mathcal{I}(\bar{z})$ as well as $\mathcal{H}(\bar{y})$.

   b. The second element is the (n+1)-postfix message type $\mathcal{H}\lfloor W_R$, where $W_R = \{\bar{y}' | \langle x', \bar{y}', \bar{z}'\rangle \in T'\}$;

   c. The third element is the (m+1)-postfix message type $\mathcal{I}\lfloor W_S$, where $W_S = \{\bar{z}' | \langle x', \bar{y}', \bar{z}'\rangle \in T'\}$;

   d. The fourth element is $(\mathcal{J}\sharp\mathcal{H}_n)\lceil V_R$, where $V_R = \{\bar{y} | \langle x, \bar{y}, \bar{z}\rangle \in T\}$;

   e. The fifth element is $(\mathcal{K}\sharp\mathcal{I}_m)\lceil V_S$, where $V_S = \{\bar{z} | \langle x, \bar{y}, \bar{z}\rangle \in T\}$;

   f. The sixth element is $max(\{P(\mathcal{H}_n(\bar{y}) \cap \mathcal{I}_m(\bar{z}) \neq \phi \mid \exists x s.t. (x, \bar{y}, \bar{z}) \in T)\}) \cdot q$, and;

   g. The seventh element is $C \cup \{c_1\}$, where c_1 is the constraint $\text{length}(\mathcal{H}_n) = \text{length}(\mathcal{I}_n)$.

   We call this first node the *node generated by the constraint* $\text{length}(\mathcal{H}_n) = \text{length}(\mathcal{I}_m)$.

2. The second node corresponds to the case in which a type from $\mathcal{H}$ can be confused wit prefix of a type from $\mathcal{I}$.

   Let T be the set of all $\langle x, \bar{y}, \bar{z}\rangle$ such that $P(\mathcal{H}_n(\bar{y}) \sqcap Pre(\mathcal{I}_m)(\bar{z})) \cdot q > p$. Then, if T is non-empty, we construct a child node as follows:

   a. The first element of the new tuple is the set T' of all $\langle x', \bar{y}', \bar{z}'\rangle$ such that there exists $\langle x, \bar{y}, \bar{z}\rangle \in T$ such that $x' = x||y_1$, where $y_1 \in \mathcal{H}_n(\bar{y})$, $\bar{y}' = append(\bar{y}.\langle y_1\rangle)$, and $\bar{z}' = append(\bar{z}, \langle y_1\rangle)$;

   Note that, in this case y_1 is an element of $Pre(\mathcal{I}_m)(\bar{z}))$ as well.

   b. The second element is the (n+1)-postfix message type $\mathcal{H}\lfloor W_R$, where $W_R = \{\bar{y}' | \langle x', \bar{y}', \bar{z}'\rangle \in T'\}$;

   c. The third element is the m-postfix message type $Split(\mathcal{I})\lfloor W_S$, where $W_S = \{\bar{z}' | \langle x', \bar{y}', \bar{z}'\rangle \in T'\}$;

   d. The fourth element is $(\mathcal{J}\sharp\mathcal{H}_n)\lceil V_R$, where $V_R = \{\bar{y} | \langle x, \bar{y}, \bar{z}\rangle \in T\}$;

   e. The fifth element is $(\mathcal{K}\sharp Pre(\mathcal{I}_m))\lceil V_S$, where $V_S = \{\bar{z} | \langle x, \bar{y}, \bar{z}\rangle \in T\}$;

   f. The sixth element of the tuple is $max(\{P(\mathcal{H}_n(\bar{y}) \sqcap Pre(\mathcal{I}_m)(\bar{z}) \mid \exists x s.t. (x, \bar{y}, \bar{z}) \in T))\}) \cdot q$, and;

   g. The seventh element is $C \cup \{c_1\}$, where c_1 is the constraint $\text{length}(\mathcal{H}_n) < \text{length}(\mathcal{I}_m)$.

   We call this node the *node generated by the constraint* $\text{length}(\mathcal{H}_n) < \text{length}(\mathcal{I}_m)$.

3. The third node corresponds to the case in which a prefix of a type from $\mathcal{H}$ can be confused with a type from $\mathcal{I}$.

   Let T be the set of all $\langle x, \bar{y}, \bar{z}\rangle$ in Usuch that $P(Pre(\mathcal{H}_n)(\bar{y}) \sqcap \mathcal{I}(\bar{z})) \cdot q > p$. Then, if T is nonempty, we construct a child node as follows:

   a. The first element of the new tuple is the set T' of all $\langle x', \bar{y}', \bar{z}'\rangle$ such that there exists $\langle x, \bar{y}, \bar{z}\rangle \in T$ such that $x' = x||y_1$, where $y_1 \in Pre(\mathcal{H}_n)(\bar{y})$, $\bar{y}' = append(\bar{y}, \langle y_1\rangle)$, and $\bar{z}' = append(\bar{z}, \langle y_1\rangle)$;

   Note that, in this case y_1 is an element $\mathcal{I}_m(\bar{z})$ as well.

   b. The second element is the n-postfix message type $Split(\mathcal{H})\lfloor W_R$, where $W_R = \{\bar{y}' | \langle x', \bar{y}', \bar{z}'\rangle \in T'\}$;

   c. The third element is the (m+1)-postfix message type $\mathcal{I}\lfloor W_S$, where $W_S = \{\bar{z}' | \langle x', \bar{y}', \bar{z}'\rangle \in T'\}$;

   d. The fourth element is $(\mathcal{J}\sharp Pre(\mathcal{H}_n)))\lceil V_R$, where $V_R = \{\bar{y} | \langle x, \bar{y}, \bar{z}\rangle \in T\}$;

   e. The fifth element is $(\mathcal{K}\sharp\mathcal{I}_m)\lceil V_S$, where $V_S = \{\bar{z} | \langle x, \bar{y}, \bar{z}\rangle \in T\}$;

   f. The sixth element is $max(\{P(Pre(\mathcal{H}_n)(\bar{y}) \sqcap \mathcal{I}_m(\bar{z})) \mid \exists x s.t. (x, \bar{y}, \bar{z}) \in T)\}) \cdot q$, and;

   g. The seventh element is $C \cup \{c_1\}$, where c_1 is the constraint $\text{length}(\mathcal{H}_n) > \text{length}(\mathcal{I}_m)$.

We call this node the *node generated by the constraint* $\text{length}(\mathcal{H}_n) > \text{length}(\mathcal{I}_m)$.

The idea behind the nodes in the tree is as follows. The first entry in the sept-tuple corresponds to the part of the zipper that we have found so far. The second and third corresponds to the portions of $\mathcal{R}$ and $\mathcal{S}$ that are still to be compared. The fourth and fifth correspond to the portions of $\mathcal{R}$ and $\mathcal{S}$ that we have compared so far. The sixth entry gives an upper bound on the probability that this portion of the zipper can be constructed by an attacker. The seventh entry gives the constraints on lengths of fields that are satisfied by this portion of the zipper.

Definition 5.1 *We say that a zipper* succeeds *if it contains a node* $\langle U, \langle\rangle, \langle\rangle, \mathcal{J}, \mathcal{K}, q, C\rangle$.

Theorem 5.1 *The zipper terminates for bounded message types, and, whether or not it terminates, it succeeds if there are any type confusions of probability greater than p. For bounded message types, the complexity is exponential in the number of message fields.*

6 An Example: An Analysis of GDOI

In this section we give a partial analysis of the signed messages of a simplified version of the GDOI protocol.

There are actually three such messages. They are: the POP signed by the group member, the POP signed by the GCKS, and the Groupkey Push Message signed by the GCKS. We will show how the POP signed by the GCKS can be confused with the Groupkey Push Message.

The POP is of the form $NONCE_A, NONCE_B$ where $NONCE_A$ is a random number generated by a group member, and $NONCE_B$ is a random number generated by the GCKS. The lengths of $NONCE_A$ and $NONCE_B$ are not constrained by the protocol. Since we are interested in the types local to the GCKS, we have $NONCE_A$ the type consisting of all numbers, and $NONCE_B$ the type local to the GCKS consisting of the the single nonce generated by the GCKS.

We can thus define the POP as a message type local to the GCKS as follows:

1. $\mathcal{R}(\langle\rangle) = NONCE_A$ where $NONCE_A$ is the type under the control of the intruder consisting of all numbers, and;
2. $\mathcal{R}(\langle y_1\rangle) = NONCE_B$ where $NONCE_B$ is a type under control of the GCKS.

We next give a simplified (for the purpose of exposition) Groupkey Push Message. We describe a version that consists only of the Header and the Key Download Payload:

$NONCE_H, \text{kd}, MESSAGE\_LENGTH, \text{sig},$
$KDLENGTH, KDHEADER, KEYS$

The $NONCE_H$ at the beginning of the header is of fixed length (16 bytes). The one-byte kd field gives the type of the first payload, while the 4-byte $MESSAGE\_LENGTH$ gives the length of the message in bytes. The one-byte sig field gives the type of the next payload (in this case the signature, which is not part of the signed message), while the 2-byte $KDLENGTH$ gives the length of the key download payload. We divide the key download data into two parts, a header which gives information about the keys, and the key data, which is random and controlled by the GCKS. (This last is greatly simplified from the actual GDOI specification).

We can thus define the Groupkey Push Message as the following message type local to the intended receiver:

1. $\mathcal{S}(\langle\rangle) = NONCE_H$ where $NONCE_H$ is the type consisting of all 16-byte numbers;
2. $\mathcal{S}(\langle x_1\rangle) = \{\text{kd}\}$;
3. $\mathcal{S}(\langle x_1, x_2\rangle) = MESSAGE\_LENGTH$, where $MESSAGE\_LENGTH$ is the type consisting of all 4-byte numbers;
4. $\mathcal{S}(\langle x_1, x_2, x_3\rangle) = \{\text{sig}\}$;
5. $\mathcal{S}(\langle x_1, x_2, x_3, x_4\rangle) = KDLENGTH$, where $KDLENGTH$ is the type consisting of all 2-byte numbers;
6. $\mathcal{S}(\langle x_1, x_2, x_3, x_4, x_5\rangle) = KDHEADER$, where the type $KDHEADER$ consists of all possible KD headers whose length is less than $x_3 - \text{length}(x_1||x_2||x_3||x_4||x_5)$ and the value of x_5.
7. $\mathcal{S}(\langle x_1, x_2, x_3, x_4, x_5, x_6\rangle) = KEYS$, where $KEYS$ is the set of all numbers whose length is less than $x_3 - \text{length}(x_1||x_2||x_3||x_4||x_5||x_6)$ and equal to $x_5 - \text{length}(x_6)$. Note that the second constraint makes the first redundant.

All of the above types are local to the receiver, but under the control of the sender.

We begin by creating the first three child nodes. All three cases length(y_1) = length(x_1), length(y_1) < length(x_1), and length(y_1) > length(x_1), are non-trivial, since $x_1 \in NONCE_H$ is an arbitrary 16-byte number, and $y_1 \in NONCE_A$ is a completely arbitrary number. Hence the probability of $NONCE_A \sqcap NONCE_B$ is one in all cases. But let's look at the children of these nodes. For the node corresponding to length(y_1) = length(x_1), we need to compare x_2 and y_2. The term x_2 is the payload identifier corresponding to "kd". It is one byte long. The term y_2 is the random nonce $NONCE_B$ generated by the GCKS. Since y_2 is the last field in the POP, there is only one possibility; that is, length(x_2) < length(y_2).

But this would require a member of $Pre(NONCE_B)$ to be equal to "kd". Since $NONCE_B$ is local to the GCKS and under its control, the chance of this is $1/2^8$. If this is not too small to worry about, we construct the child of this node. Again, there will be only one, and it will correspond to length(x_3) < length(y_2) - length(x_2). In this case, x_3 is the apparently arbitrary number $MESSAGE\_LENGTH$. But there is a nontrivial relationship between $MESSAGE\_LENGTH$ and $NONCE_B$, in that $MESSAGE\_LENGTH$ must describe a length equal to $M + N$, where M is the length of the part of $NONCE_B$ remaining after the point at which $MESSAGE\_LENGTH$ appears in it, and N describes the length of the signature payload. Since both of these lengths are outside of the intruder's control, the probability that the first part of $NONCE_B$ will have exactly this value is $1/2^{16}$. We are now up to a probability of $1/2^{24}$.

When we go to the next child node, again the only possibility is length(x_4) < length(y_2) - length(x_3) - length(x_2), and the comparison in this case is with the 1-byte representation of "sig". The probability of type confusion now becomes $1/2^{32}$. If this is still a concern, we can continue in this fashion, comparing pieces of $NONCE_B$ with the components of the Groupkey Push Message until the risk has been reduced to an acceptable level. A similar line of reasoning works for the case length(y_1) < length(x_1).

We now look at the case length(y_1) > length(x_1), and show how it can be used to construct the attack we mentioned at the beginning of this paper. We concentrate on the child node generated by the constraint length(y_1) - length(x_1) > length(x_2). Since $y_1 \in NONCE_A$ is an arbitrary number, the probability that x_2 can be taken for a piece of y_1, given the length constraint, is one. We continue in this fashion, until we come to the node generated by the constraint length(x_7) <length(y_1) - $\sum_{i=1}^{5} x_i$. The remaining field of the Groupkey Pull Message, $x_7 \in KEYS$ is an arbitrary number, so the chance that the remaining field of the POP, y_2 together with what remains of y_1, can be mistaken for x_7, is one, since the concatenation of the remains of y_1 with y_2, by definition, will be a member of the abitrary set $KEYS$.

7 Conclusion and Discussion

We have developed a procedure for determining whether or not type confusions are possible in signed messages in a cryptographic protocol. Our approach has certain advantages over previous applications of formal methods to type confusion; we can take into account the possibility that an attacker could cause pieces of message fields to be confused with each other, as well as entire fields. It also takes into account the probability of an attack succeeding. Thus, for example, it would catch message type attacks in which typing tags, although present, are so short that it is possible to generate them randomly with a non-trivial probability.

Our greater generality comes at a cost, however. Our procedure is not guaranteed to terminate for unbounded message types, and even for bounded types it is exponential in the number of message fields. Thus, it would have not have terminated for the actual, unsimplified, GDOI protocol, which allows an arbitrary number of keys in the Key Download payload, although it still would have found the type confusion attacks that we described at the beginning of this paper.

Also, we have left open the problem of how the probabilities are actually computed, although in many cases, such as that of determining whether or not a random number can be mistaken for a formatted field, this is fairly straightforward. In other cases, as in the comparison between $NONCE_B$ and $MESSAGE\_LENGTH$ from above, things may be more tricky. This is because, even though the type of a field is a function of the fields that come before it in a message, the values of the fields that come after it may also act as a constraint, as the length of the part of the message appearing after $MESSAGE\_LENGTH$ does on the value of $MESSAGE\_LENGTH$.

Other subtleties may arise from the fact that other information that may or may not be available to the intruder may affect the probability of type confusion. For example, in the comparison between $MESSAGE\_LENGTH$ and $NONCE_B$, the intruder has to generate $NONCE_A$ before it sees $NONCE_B$. If it could generate $NONCE_A$ after it saw $NONCE_B$, this would give it some more control over the placement of $MESSAGE\_LENGTH$ with respectt to $NONCE_B$. This would increase the likelyhood that it would be able to force $MESSAGE\_LENGTH$ to have the appropriate value.

But, although we will need to deal with special cases like these, we believe that, in practice, the number of different types of such special cases will be small, and thus we believe that it should be possible to narrow the problem down so that a more efficient and easily automatable approach becomes possible. In particular, a study of the most popular approaches to formatting cryptographic protocols should yield some insights here.

8 Acknowledgements

We are greatful to MSec and SMuG Working Groups, and in particular to the authors for the GDOI protocol, for many helpful discussions on this topic. This work was supported by ONR.

References

[1] J. Alves-Foss. Provably insecure mutual authentication protocols: The two party symmetric encryption

case. In *Proc. 22nd National Information Systems Security Conference.*, Arlington, VA, 1999.

[2] Mark Baugher, Thomas Hardjono, Hugh Harney, and Brian Weis. The Group Domain of Interpretation. Internet Draft draft-ietf-msec-gdoi-04.txt, Internet Engineering Task Force, February 26 2002. available at http://www.ietf.org/internet-drafts/draft-ietf-msec-gdoi-04.txt.

[3] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). RFC 2409, Internet Engineering Task FOrce, November 1998. available at http://ietf.org/rfc/rfc2409.txt.

[4] James Heather, Gavin Lowe, and Steve Schneider. How to prevent type flaw attacks on security protocols. In *Proceedings of 13th IEEE Computer Security Foundations Workshop*, pages 255–268. IEEE Computer Society Press, June 2000. A revised version is to appear in the *Journal of Computer Security*.

[5] John Kelsey and Bruce Schneier. Chosen interactions and the chosen protocol attack. In *Security Protocols, 5th International Workshop April 1997 Proceedings*, pages 91–104. Springer-Verlag, 1998.

[6] D. Maughan, M. Schertler, M. Schneider, and J. Turner. Internet Security Association and Key Management Protocol (ISAKMP). Request for Comments 2408, Network Working Group, November 1998. Available at http://ietf.org/rfc/rfc2408.txt.

[7] Catherine Meadows. Analyzing the Needham-Schroeder public key protocol: A comparison of two approaches. In *Proceedings of ESORICS '96*. Springer-Verlag, 1996.

[8] Einar Snekkenes. Roles in cryptographic protocols. In *Proceedings of the 1992 IEEE Computer Security Symposium on Research in Security and Privacy*, pages 105–119. IEEE Computer Society Press, May 4-6 1992.

Finding Counterexamples to Inductive Conjectures and Discovering Security Protocol Attacks

Graham Steel, Alan Bundy, and Ewen Denney
Division of Informatics
University of Edinburgh

{grahams,bundy,ewd}@dai.ed.ac.uk

Abstract

We present an implementation of a method for finding counterexamples to universally quantified conjectures in first-order logic. Our method uses the proof by consistency strategy to guide a search for a counterexample and a standard first-order theorem prover to perform a concurrent check for inconsistency. We explain briefly the theory behind the method, describe our implementation, and evaluate results achieved on a variety of incorrect conjectures from various sources.

Some work in progress is also presented: we are applying the method to the verification of cryptographic security protocols. In this context, a counterexample to a security property can indicate an attack on the protocol, and our method extracts the trace of messages exchanged in order to effect this attack. This application demonstrates the advantages of the method, in that quite complex side conditions decide whether a particular sequence of messages is possible. Using a theorem prover provides a natural way of dealing with this. Some early results are presented and we discuss future work.

1 Introduction

Inductive theorem provers are frequently employed in the verification of programs, algorithms and protocols. However, programs and algorithms often contain bugs, and protocols may be flawed, causing the proof attempt to fail. It can be hard to interpret a failed proof attempt: it may be that some additional lemmas need to be proved or a generalisation made. In this situation, a tool which can not only detect an incorrect conjecture, but also supply a counterexample in order to allow the user to identify the bug or flaw, is potentially very valuable. The problem of cryptographic security protocol verification is a specific area in which incorrect conjectures are of great consequence. If a security conjecture turns out to be false, this can indicate an attack on the protocol. A counterexample can help the user to see how the protocol can be attacked. Incorrect conjectures also arise in automatic inductive theorem provers where generalisations are speculated by the system. Often we encounter the problem of over-generalisation: the speculated formula is not a theorem. A method for detecting these over-generalisations is required.

Proof by consistency is a technique for automating inductive proofs in first-order logic. Originally developed to prove correct theorems, this technique has the property of being refutation complete, i.e. it is able to refute in finite time conjectures which are inconsistent with the set of hypotheses. When originally proposed, this technique was of limited applicability. Recently, Comon and Nieuwenhuis have drawn together and extended previous research to show how it may be more generally applied, [10]. They describe an experimental implementation of the inductive completion part of the system. However, the check for refutation or consistency was not implemented. This check is necessary in order to ensure a theorem is correct, and to automatically refute an incorrect conjecture. We have implemented a novel system integrating Comon and Nieuwenhuis' experimental prover with a concurrent check for inconsistency. By carrying out the check in parallel, we are able to refute incorrect conjectures in cases where the inductive completion process fails to terminate. The parallel processes communicate via sockets using Linda, [8].

The ability of the technique to prove complex inductive theorems is as yet unproven. That does not concern us here – we are concerned to show that it provides an efficient and effective method for refuting *incorrect* conjectures. However, the ability to prove at least many small theorems helps alleviate a problem reported in Protzen's work on disproving conjectures, [22] – that the system terminates only at its depth limit in the case of a small unsatisfiable formula, leaving the user or proving system none the wiser.

We have some early results from our work in progress, which is to apply the technique to the aforementioned problem of cryptographic security protocol verification. These protocols often have subtle flaws in them that are not detected for years after they have been proposed. By

devising a first-order version of Paulson's inductive formalism for the protocol verification problem, [21], and applying our refutation system, we can not only detect flaws but also automatically generate the sequence of messages needed to expose these flaws. By using an inductive model with arbitrary numbers of agents and runs rather than the finite models used in most model-checking methods, we have the potential to synthesise parallel session and replay attacks where a single principal may be required to play multiple roles in the exchange.

In the rest of the paper, we first review the literature related to the refutation of incorrect conjectures and proof by consistency, then we briefly examine the Comon-Nieuwenhuis method. We describe the operation of the system, relating it to the theory, and present and evaluate the results obtained so far. The system has been tested on a number of examples from various sources including Protzen's work [22], Reif et al.'s, [24], and some of our own. Our work in progress on the application of the system to the cryptographic protocol problem is then presented. Finally, we describe some possible further work and draw some conclusions.

2 Literature Review

2.1 Refuting Incorrect Conjectures

At the CADE-15 workshop on proof by mathematical induction, it was agreed that the community should address the issue of dealing with non-theorems as well as theorems[1]. However, relatively little work on the problem has since appeared. In the early nineties Protzen presented a sound and complete calculus for the refutation of faulty conjectures in theories with free constructors and complete recursive definitions, [22]. The search for the counterexample is guided by the recursive definitions of the function symbols in the conjecture. A depth limit ensures termination when no counterexample can be found.

More recently, Reif et al., [25], have implemented a method for counterexample construction that is integrated with the interactive theorem prover KIV, [23]. Their method incrementally instantiates a formula with constructor terms and evaluates the formulae produced using the simplifier rules made available to the system during proof attempts. A heuristic strategy guides the search through the resulting subgoals for one that can be reduced to $false$. If such a subgoal is not found, the search terminates when all variables of generated sorts have been instantiated to constructor terms. In this case the user is left with a model condition, which must be used to decide whether the instantiation found is a valid counterexample.

Ahrendt has proposed a refutation method using model construction techniques, [1]. This is restricted to free datatypes, and involves the construction of a set of suitable clauses to send to a model generation prover. As first reported, the approach was not able in general to find a refutation in finite time, but new work aims to address this problem, [2].

2.2 Proof by Consistency

Proof by consistency is a technique for automating inductive proof. It has also been called *inductionless induction*, and *implicit induction*, as the actual induction rule used is described implicitly inside a proof of the conjecture's consistency with the set of hypotheses. Recent versions of the technique have been shown to be *refutation complete*, i.e. are guaranteed to detect non-theorems in finite time.[2] The proof by consistency technique was developed to solve problems in equational theories, involving a set of equations defining the *initial model*[3], E. The first version of the technique was proposed by Musser, [20], for equational theories with a *completely defined equality predicate*, This requirement placed a strong restriction on the applicability of the method. The completion process used to deduce consistency was the Knuth-Bendix algorithm, [17].

Huet and Hullot [14] extended the method to theories with free constructors, and Jouannaud and Kounalis, [15], extended it further, requiring that E should be a convergent rewrite system. Bachmair, [4], proposed the first refutationally complete deduction system for the problem, using a *linear strategy* for inductive completion. This is a restriction of the Knuth-Bendix algorithm which entails only examining overlaps between axioms and conjectures. The key advantage of the restricted completion procedure was its ability to cope with unoriented equations. The refutational completeness of the procedure was a direct result of this.

The technique has been extended to the non-equational case. Ganzinger and Stuber, [12], proposed a method for proving consistency for a set of first-order clauses with equality using a refutation complete linear system. Kounalis and Rusinowitch, [18], proposed an extension to conditional theories, laying the foundations for the method implemented in the SPIKE theorem, [6]. Ideas from the proof by consistency technique have been used in other induction methods, such as cover set induction, [13], and test set induction, [5].

[1] The minutes of the discussion are available from `http://www.cee.hw.ac.uk/~air/cade15/cade-15-mind-ws-session-3.html`.

[2] Such a technique must necessarily be incomplete with respect to proving theorems correct, by Gödel's incompleteness theorem.

[3] The initial or standard model is the minimal Herbrand model. This is unique in the case of a purely equational specification.

2.3 Cryptographic Security Protocols

Cryptographic protocols are used in distributed systems to allow agents to communicate securely. Assumed to be present in the system is a spy, who can see all the traffic in the network and may send malicious messages in order to try and impersonate users and gain access to secrets. Clark and Jacob's survey, [9], and Anderson and Needham's article, [3], are good introductions to the field.

Although security protocols are usually quite short, typically 2–5 messages, they often have subtle flaws in them that may not be discovered for many years. Researchers have applied various formal methods techniques to the problem, to try to find attacks on faulty protocols, and to prove correct protocols secure. These approaches include belief logics such as the so-called BAN logic, [7], state-machines, [11, 16], model-checking, [19], and inductive theorem proving, [21]. Each approach has its advantages and disadvantages. For example, the BAN logic is attractively simple, and has found some protocol flaws, but has missed others. The model checking approach can find flaws very quickly, but can only be applied to finite (and typically very small) instances of the protocol. This means that if no attack is found, there may still be an attack upon a larger instance. Modern state machine approaches can also find and exhibit attacks quickly, but require the user to choose and prove lemmas in order to reduce the problem to a tractable finite search space. The inductive method deals directly with the infinite state problem, and assumes an arbitrary number of protocol participants, but proofs are tricky and require days or weeks of expert effort. If a proof breaks down, there are no automated facilities for the detection of an attack.

3 The Comon-Nieuwenhuis Method

Comon and Nieuwenhuis, [10], have shown that the previous techniques for proof by consistency can be generalised to the production of a first-order axiomatisation $\mathcal{A}$ of the minimal Herbrand model such that $\mathcal{A} \cup E \cup C$ is consistent if and only if C is an inductive consequence of E. With $\mathcal{A}$ satisfying the properties they define as an *I-Axiomatisation*, inductive proofs can be reduced to first-order consistency problems and so can be solved by any saturation based theorem prover. We give a very brief summary of their results here. Suppose I is, in the case of Horn or equational theories, the unique minimal Herbrand model, or in the case of non-Horn theories, the so-called *perfect model* with respect to a total ordering on terms, $\succ$[4]:

Definition 1 *A set of first-order formulae $\mathcal{A}$ is an* I-Axiomatisation *of I if*

1. *$\mathcal{A}$ is a set of purely universally quantified formulae*
2. *I is the only Herbrand model of $E \cup \mathcal{A}$ up to isomorphism.*

An I-Axiomatisation is normal *if $\mathcal{A} \models s \neq t$ for all pairs of distinct normal terms s and t*

The I-Axiomatisation approach produces a clean separation between the parts of the system concerned with inductive completion and inconsistency detection. Completion is carried out by a saturation based theorem prover, with inference steps restricted to those produced by conjecture superposition, a restriction of the standard superposition rule. Only overlaps between conjecture clauses and axioms are considered. Each non-redundant clause derived is checked for consistency against the I-Axiomatisation. If the theorem prover terminates with saturation, the set of formulae produced comprise a *fair induction derivation*. The key result of the theory is this:

Theorem 1 *Let $\mathcal{A}$ be a normal I-Axiomatisation, and $C_0, C_1, \ldots$ be a fair induction derivation. Then $I \models C_0$ iff $\mathcal{A} \cup \{c\}$ is consistent for all clauses c in $\bigcup_i C_i$.*

This theorem is proved in [10]. Comon and Nieuwenhuis have shown that this conception of proof by consistency generalises and extends earlier approaches. An equality predicate as defined by Musser, a set of free constructors as proposed by Huet and Hullot or a ground reducibility predicate as defined by Jouannaud and Kounalis could all be used to form a suitable I-Axiomatisation. The technique is also extended beyond ground convergent specifications (equivalent to saturated specifications for first-order clauses) as required in [15, 4, 12]. Previous methods, e.g. [6], have relaxed this condition by using conditional equations. However a ground convergent rewrite system was still required for deducing inconsistency. Using the I-Axiomatisation method, conjectures can be proved or refuted in (possibly non-free) constructor theories which cannot be specified by a convergent rewrite system.

Whether these extensions to the theory allow larger theorems to be *proved* remains to be seen, and is not of interest to us here. We are interested in how the wider applicability of the method can allow us to investigate the ability of the proof by consistency technique to root out a counterexample to realistic *incorrect* conjectures.

4 Implementation

Figure 1 illustrates the operation of our system. The input is an inductive problem in `Saturate` format and a normal I-Axiomatisation (see Definition 1, above). The version of `Saturate` customised by Nieuwenhuis for implicit induction (the right hand box in the diagram) gets the problem file only, and proceeds to pursue inductive completion, i.e. to derive a fair induction derivation. Every

[4] Saturation style theorem proving always requires that we have such an ordering available.

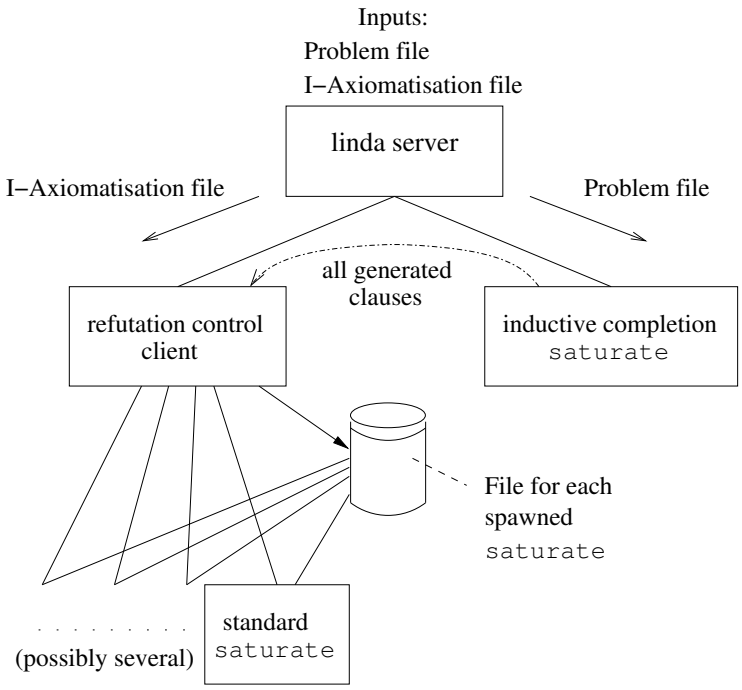


Figure 1: System operation

non-redundant clause generated is passed via the server to the refutation control program (the leftmost box). For every new clause received, this program generates a problem file containing the I-Axiomatisation and the new clause, and spawns a standard version of `Saturate` to check the consistency of the file. Crucially, these spawned `Saturates` are not given the original axioms – only the I-Axioms are required, by Theorem 1. This means that almost all of the search for an inconsistency is done by the prover designed for inductive problems and the spawned `Saturates` are just used to check for inconsistencies between the new clauses and the I-Axiomatisation. This should lead to a false conjecture being refuted after fewer inference steps have been attempted than if the conjecture had been given to a standard first-order prover together with all the axioms and I-Axioms. We evaluate this in the next section.

If, at any time, a refutation is found by a spawned prover, the proof is written to a file and the completion process and all the other spawned `Saturate` processes are killed. If completion is reached by the induction prover, this is communicated to the refutation control program, which will then wait for the results from the spawned processes. If they all terminate with saturation, then there are no inconsistencies, and so the theorem has been proved (by Theorem 1).

There are several advantages to the parallel architecture we have employed. One is that it allows us to refute incorrect conjectures even if the inductive completion process does not terminate. This would also be possible by modifying the main induction `Saturate` to check each clause in tern, but this would result in a rather messy and unwieldy program. Another advantage is that we are able to easily devote a machine solely to inductive completion in the case of harder problems. It is also very convenient when testing a new model to be able to just look at the deduction process before adding the consistency check later on, and we preserve the attractive separation in the theory between the deduction and the consistency checking processes.

A disadvantage of our implementation is that launching a new `Saturate` process to check each clause against the I-Axiomatisation generates some overheads in terms of disk access etc. In our next implementation, when a spawned prover reaches saturation (i.e. no inconsistencies), it will clear its database and ask the refutation control client for another clause to check, using the existing sockets mechanism. This will cut down the amount of memory and disk access required. A further way to reduce the consistency checking burden is to take advantage of knowledge about the structure of the I-Axiomatisation for simple cases. For example, in the case of a free constructor specification, the I-Axiomatisation will consist of clauses specifying the inequality of non-identical constructor terms. Since it will include no rules referring to defined symbols, it is sufficient to limit the consistency check to generated clauses containing only constructors and variables.

Table 1: Sample of results. *In the third column, the first number shows the number of clauses derived by the inductive completion prover, and the number in brackets indicates the number of clauses derived by the parallel checker to spot the inconsistency. The fourth column shows the number of clauses derived by an unmodified first-order prover when given the conjecture, axioms and I-Axioms all together.*

Problem	Counterexample found	No. of clauses derived to find refutation	No. of clauses derived by a standard prover
$\forall N, M. \neg(s(N) + M = s(0))$	$N = 0, M = 0$	2(+0)	2
$X \neq Y \wedge X \neq 0 \wedge Y \neq 0$ $\Rightarrow (X \geq Y \wedge Y \neq 0) \vee$ $(X \neq 0 \wedge Y = 0)$	$X = s(0)$, $Y = s(s(X))$	4 (+3)	6
$app(K, L) = app(L, K)$	$K = 0, L = s(X)$	9(+11)	stuck in loop
$sort(l_1) \wedge l_2 = ap(l_1, [head(l_3)])$ $\wedge length(l_3) \geq 2 * length(l_1)$ $\wedge l_3 \neq nil \wedge$ $member(head(l_1), tail(l_3))$ $\Rightarrow sort(l_2)$	$l_1 = [s(X)]$, $l_2 = [s(X), 0]$ $l_3 = [0, s(X)\|Y]$	55(+1)	76
All graphs are acyclic	$[e(a, a)]$	99	123
All loopless graphs are acyclic	$[e(s(a), a), e(a, s(a))]$	178	2577
$gcd(X, X) = 0$	$X = s(0)$	17(+2)	29
Impossibility property for Neuman-Stubblefield key exchange protocol	$[msg(1), msg(2)$, $msg(3), msg(4)]$	866 (+0)	1733
Authenticity property for simple protocol from [9]	see section 6	730(+1)	3148

5 Evaluation of Results

Table 1 shows a sample of results achieved so far. The first three examples are from Protzen's work, [22], the next two from Reif et al.'s, [25], and the last three are from our work. The gcd example is included because previous methods of proof by consistency could not refute this conjecture. Comon and Nieuwenhuis showed how it could be tackled, [10], and here we confirm that their method works. The last two conjectures are about properties of security protocols. The 'impossibility property' states that no trace reaches the end of a protocol. Its refutation comprises the proof of a possibility property, which is the first thing proved about a newly modelled protocol in Paulson's method, [21]. The last result is the refutation of an authenticity property, indicating an attack on the protocol. This protocol is a simple example included in Clark's survey, [9], for didactic purposes, but requires that one principal play both roles in a protocol run. More details are given in section 6.

Our results on Reif et al.'s examples do not require the user to verify a model condition, as the system described in their work does. Interestingly, the formula remaining as a model condition in their runs is often the same as the formula which gives rise to the inconsistency when checked against the I-Axiomatisation in our runs. This is because the KIV system stops when it derives a term containing just constructors and variables. In such a case, our I-Axiomatisation would consist of formulae designed to check validity of these terms. This suggests a way to automate the model condition check in the KIV system.

On comparing the number of clauses derived by our system and the number of clauses required by a standard first-order prover (SPASS), we can see that the proof by consistency strategy does indeed cut down on the number of inferences required. This is more evident in the larger examples. Also, the linear strategy allows us to cope with commutativity conjectures, like the third example, which cause a standard prover to go into a loop. We might ask: what elements of the proof by consistency technique are allowing us to make this saving in required inferences? One is the refutation completeness result for the linear strategy, so we know we need only consider overlaps between conjectures and axioms. Additionally, separating the I-Axioms from the theory axioms reduces the number of overlaps between conjectures and axioms to be considered each time. We also use the results about inductively complete positions for theories with free constructors, [10]. This applies to all the examples except those in graph theory, where we used Reif's formalism and hence did not have free constructors. This is the probable reason why, on these two examples, our system did not make as large a saving in

derived clauses.

The restriction to overlaps between conjectures and axioms is similar in nature to the so-called set of support strategy, using the conjecture as the initial supporting set. The restriction in our method is tighter, since we don't consider overlaps between formulae in the set of support. Using the set of support strategy with the standard prover on the examples in Table 1, refutations are found after deriving fewer clause than required by the standard strategy. However, performance is still not as good as for our system, particularly in the free constructor cases. The set of support also doesn't fix the problem of divergence on unoriented conjectures, like the commutativity example.

The efficiency of the method in terms of clauses derived compared to a standard prover looks good. However, actual time taken by our system is much longer than that for the standard SPASS. This is because the `Saturate` prover is rather old, and was not designed to be a serious tool for large scale proving. In particular, it does not utilise any term indexing techniques, and so redundancy checks are extremely slow. As an example, the impossibility property took about 50 minutes to refute in `Saturate`, but about 40 seconds in SPASS, even though more than twice as many clauses had to be derived. We used `Saturate` in our first system as Nieuwenhuis had already implemented the proof by consistency strategy in the prover. A re-implementation of the whole system using SPASS should give us even faster refutations, and is one of our next tasks.

Finally, we also tested the system on a number of small inductive theorems. Being able to prove small theorems allows us to attack a problem highlighted in Protzen's work: that if an candidate generalisation (say) is given to the counterexample finder and it returns a result saying that the depth limit was reached before a counterexample was found, the system is none the wiser as to whether the generalisation is worth pursuing. If we are able to prove at least small examples to be theorems, this will help alleviate the problem. Our results were generally good: 7 out of 8 examples we tried were proved, but one was missed. Comon intends to investigate the ability of the technique to prove more and larger theorems in future.

More details of the results including some sample runs and details of the small theorems proved can be found at `http://www.dai.ed.ac.uk/~grahams/linda`.

6 Application to Cryptographic Security Protocols

We now describe some work in progress on applying our technique to the cryptographic security protocol problem. As we saw in section 2.3, one of the main thrusts of research has been to apply formal methods to the problem. Researchers have applied techniques from model checking, theorem proving and modal logics amongst others. Much attention is paid to the modelling of the abilities of the spy in these models. However, an additional consideration is the abilities of the participants. Techniques assuming a finite model, with typically two agents playing distinct roles, often rule out the possibility of discovering a certain kind of parallel session attack, in which one participant plays both roles in the protocol. The use of an inductive model allows us to discover these kind of attacks. An inductive model also allows us to consider protocols with more than two participants, e.g. conference-key protocols.

Paulson's inductive approach has used been used to verify properties of several protocols, [21]. Protocols are formalised in typed higher-order logic as the set of all possible traces, a trace being a list of events like 'A sends message X to B'. This formalism is mechanised in the Isabelle/HOL interactive theorem prover. Properties of the security protocol can be proved by induction on traces. The model assumes an arbitrary number of agents, and any agent may take part in any number of concurrent protocol runs playing any role. Using this method, Paulson discovered a flaw in the simplified Otway-Rees shared key protocol, [7], giving rise to a parallel session attack where a single participant plays both protocol roles. However, as Paulson observed, a failed proof state can be difficult to interpret in these circumstances. Even an expert user will be unsure as to whether it is the proof attempt or the conjecture which is at fault. By applying our counterexample finder to these problems, we can automatically detect and present attacks when they exist.

Paulson's formalism is in higher-order logic. However, no 'fundamentally' higher-order concepts are used – in particular there is no unification of higher-order objects. Objects have types, and sets and lists are used. All this can be modelled in first-order logic. The security protocol problem has been modelled in first-order logic before, e.g. by Weidenbach, [26]. This model assumed a two agent model with just one available nonce[5] and key, and so could not detect the kind of parallel session attacks described. Our model allows an arbitrary number of agents to participate, playing either role, and using an arbitrary number of fresh nonces and keys.

6.1 Our Protocol Model

Our models aims to be as close as possible to a first-order version of Paulson's formalism. As in Paulson's model, agents, nonces and messages are free data types. This allows us to define a two-valued function eq which will tell us whether two pure constructor terms are equal or not. Since the rules defining eq are exhaustive, they also have the effect of suggesting instantiations where certain conditions must be met, e.g. if we require the identities of two agents to be distinct. The model is kept Horn by defin-

[5] A nonce is a unique identifying number.

ing two-valued functions for checking the side conditions for a message to be sent, e.g. we define conditions for $member(X, L) = true$ and $member(X, L) = false$ using our eq function. This cuts down the branching rate of the search.

The intruder's knowledge is specified in terms of sets. Given a trace of messages exchanged, XT, we define $analz(XT)$ to be the least set including XT closed under projection and decryption by known keys. This is accomplished by using exactly the same rules as the Paulson model, [21, p. 12]. Then, we can define the messages the intruder may send, given a trace XT, as being members of the set $synth(analz(XT))$, where $synth(X)$ is the least set including agent names closed under pairing and encryption by known keys. Again this set is defined in our model with the same axioms that Paulson uses.

A trace of messages is modelled as a list. For a specific protocol, we generally require one axiom for each protocol message. These axioms take the form of rules with the informal interpretation, 'if XT is a trace containing message n addressed to agent xa, then the trace may be extended by xa responding with message $n + 1$'. Once again, this is very similar to the Paulson model.

An example illustrates some of these ideas. In Figure 2 we demonstrate the formalism of a very simple protocol included in Clark and Jacob's survey to demonstrate parallel session attacks, [9]. Although simple, the attack on the protocol does require principal A to play the role of both initiator and responder. It assumes that A and B already share a secure key, K_{AB}. N_A denotes a nonce generated by A.

In a symmetric key protocol, principals should respond to $key(A, B)$ and $key(B, A)$, as they are in reality the same. At the moment we model this with two possible rules for message 2, but it should be straightforward to extend the model to give a cleaner treatment of symmetric keys as sets of agents. Notice we allow a principal to respond many times to the same message, as Paulson's formalism does.

The second box, Figure 3, shows how the refutation of a conjectured security property leads to the discovery of the known attack. At the moment, choosing which conjectures to attempt to prove is tricky. A little thought is required in order to ensure that only a genuine attack can refute the conjecture. More details of our model for the problem, including the specification of intruder knowledge, can be found at `http://www.dai.ed.ac.uk/~grahams/linda`.

This application highlights a strength of our refutation system: in order to produce a backwards style proof, as Paulson's system does, we must apply rules with side conditions referring as yet uninstantiated variables. For example, a rule might be applied with the informal interpretation, 'if the spy can extract X from the trace of messages sent up to this point, then he can break the security conjecture'. At the time the rule is applied, X will be uninstantiated. Further rules instantiate parts of the trace, and side conditions are either satisfied and eliminated, or found to be unsatisfiable, causing the clauses containing the condition to be pruned off as redundant. The side conditions influence the path taken through the search space, as smaller formulae are preferred by the default heuristic in the prover. This means that some traces a naïve counterexample search might find are not so attractive to our system, e.g. a trace which starts with several principals sending message 1 to other principals. This will not be pursued at first, as all the unsatisfied side conditions will make this formula larger than others.

7 Further Work

Our first priority is to re-implement the system using SPASS, and then to carry out further experiments with larger false conjectures and more complex security protocols. This will allow us to evaluate the technique more thoroughly. A first goal is to rediscover the parallel session attack discovered by Paulson. The system should also be able to discover more standard attacks, and the Clark survey, [9], provides a good set of examples for testing. We will then try the system on other protocols and look for some new attacks. A key advantage of our security model is that it allows attacks involving arbitrary numbers of participants. This should allow us to investigate the security of protocols involving many participants in a single run, e.g. conference key protocols.

In future, we also intend to implement more sophisticated heuristics to improve the search performance, utilising domain knowledge about the security protocol problem. Heuristics could include eager checks for unsatisfiable side conditions. Formulae containing these conditions could be discarded as redundant. Another idea is to vary the weight ascribed to variables and function symbols, so as to make the system inclined to check formulae with predominantly ground variables before trying ones with many uninstantiated variables. This should make paths to attacks more attractive to the search mechanism, but some careful experimentation is required to confirm this.

The Comon-Nieuwenhuis technique has some remaining restrictions on applicability, in particular the need for *reductive definitions*, a more relaxed notion of reducibility than is required for ground convergent rewrite systems. It is quite a natural requirement that recursive function definitions should be reducing in some sense. For example, the model of the security protocol problem is reductive in the sense required by Comon and Nieuwenhuis. Even so, it should be possible to extend the technique for non-theorem detection in the case of non-reductive definitions, at the price of losing any reasonable chance of proving a theorem, but maintaining the search guidance given by the proof by consistency technique. This would involve al-

The Clark-Jacob protocol demonstrating parallel session attacks. At the end of a run, A should now be assured of B's presence, and has accepted nonce N_A to identify authenticated messages.

1. $A \rightarrow B : \{\!|\ N_A |\!\}_{K_{AB}}$
2. $B \rightarrow A : \{\!|\ N_A + 1 |\!\}_{K_{AB}}$

Formula for modelling message 1 of the protocol. Informally: if XT is a trace, XA and XB agents, and XNA a number not appearing as a nonce in a previous run, then the trace may be extended by XA initiating a run, sending message 1 of the protocol to XB.

$$m(XT) = true \wedge agent(XA) = true \wedge agent(XB) = true \wedge number(XNA) = true$$
$$\wedge member(sent(X, Y, encr(nonce(XNA), K)), XT) = false \Rightarrow$$
$$m([sent(XA, XB, encr(nonce(XNA), key(XA, XB)))|XT]) = true$$

Formulae for message 2. Two formulae are used to make the response to the shared key symmetric (see text). Informally: if XT is a trace containing message 1 of the protocol addressed to agent XB, encrypted under a key he shares with agent XA, then the trace may be extended by agent XB responding with message 2.

$$member(sent(X, XB, encr(nonce(XNA), key(XA, XB))), XT) = true \wedge m(XT) = true \Rightarrow$$
$$m([sent(XB, XA, encr(s(nonce(XNA)), key(XA, XB)))|XT]) = true.$$

$$member(sent(X, XB, encr(nonce(XNA), key(XB, XA))), XT) = true \wedge m(XT) = true \Rightarrow$$
$$m([sent(XB, XA, encr(s(nonce(XNA)), key(XB, XA)))|XT]) = true.$$

Figure 2: The modelling of the Clark-Jacob protocol

The parallel session attack suggested by Clark and Jacob [9]. At the end of the attack, A believes B is operational. B may be absent or may no longer exist:

1. $A \rightarrow C_B : \{\!|\ N_A |\!\}_{K_{AB}}$
2. $C_B \rightarrow A : \{\!|\ N_A |\!\}_{K_{AB}}$
3. $A \rightarrow C_B : \{\!|\ s(N_A) |\!\}_{K_{AB}}$
4. $C_B \rightarrow A : \{\!|\ s(N_A) |\!\}_{K_{AB}}$

Below is the incorrect security conjecture, the refutation of which gives rise to the attack above. Informally this says, 'for all valid traces T, if A starts a run with B using nonce N_A, and receives the reply $s(N_A)$ from principal X, and no other principal has sent a reply, then the reply must have come from agent B.'

$$member(sent(XA, XB, encr(nonce(XNA), K), XT) = true$$
$$\wedge XT = [sent(X, XA, encr(s(nonce(XNA)), K)|T]$$
$$\wedge member(sent(Y, XA, encr(s(nonce(XNA)), K), T) = false$$
$$\wedge m(XT) = true \Rightarrow eq(X, XB) = true$$

The final line of output from the system, giving the attack.

```
c(sent(spy,a,encr(s(nonce(0)),key(a,s(a)))),
c(sent(a,s(a),encr(s(nonce(0)),key(a,s(a)))),
c(sent(spy,a,encr(nonce(0),key(a,s(a)))),
c(sent(a,s(a),encr(nonce(0),key(a,s(a)))),nil))))
```

Figure 3: The attack and its discovery

lowing inferences by standard superposition if conjecture superposition is not applicable.

8 Conclusions

In this paper we have presented a working implementation of a novel method for investigating an inductive conjecture, with a view to proving it correct or refuting it as false. We are primarily concerned with the ability of the system to refute false conjectures, and have shown results from testing on a variety of examples. These have shown that our parallel inductive completion and consistency checking system requires considerably fewer clauses to be derived than a standard first-order prover does when tackling the whole problem at once. The application of the technique to producing attacks on faulty cryptographic security protocols looks promising, and the system has already synthesised an attack of a type many finite security models will not detect. We intend to produce a faster implementation using the SPASS theorem prover, and then to pursue this application further.

References

[1] W. Ahrendt. A basis for model computation in free data types. In *CADE-17, Workshop on Model Computation - Principles, Algorithmns, Applications*, 2000.

[2] W. Ahrendt. Deductive search for errors in free data type specifications using model generation. In *CADE-18*, 2002.

[3] R. Anderson and R. Needham. *Computer Science Today: Recent Trends and Developments*, volume 1000 of *LNCS*, chapter Programming Satan's Computer, pages 426–440. Springer, 1995.

[4] L. Bachmair. *Canonical Equational Proofs*. Birkhauser, 1991.

[5] A. Bouhoula, E. Kounalis, and M. Rusinowitch. Automated mathematical induction. Rapport de Recherche 1663, INRIA, April 1992.

[6] A. Bouhoula and M. Rusinowitch. Implicit induction in conditional theories. *Journal of Automated Reasoning*, 14(2):189–235, 1995.

[7] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.

[8] N. Carreiro and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[9] J. Clark and J. Jacob. A survey of authentication protocol literature: Version 1.0. Available via `http://www.cs.york.ac.uk/jac/papers/drareview.ps.gz`, 1997.

[10] H. Comon and R. Nieuwenhuis. Induction = I-Axiomatization + First-Order Consistency. *Information and Computation*, 159(1-2):151–186, May/June 2000.

[11] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions in Information Theory*, 2(29):198–208, March 1983.

[12] H. Ganzinger and J. Stuber. *Informatik — Festschrift zum 60. Geburtstag von Günter Hotz*, chapter Inductive theorem proving by consistency for first-order clauses, pages 441–462. Teubner Verlag, 1992.

[13] M. S. Krishnamoorthy H. Zhang, D. Kapur. A mechanizable induction principle for equational specifications. In E. L. Lusk and R. A. Overbeek, editors, *Proceedings 9th International Conference on Automated Deduction, Argonne, Illinois, USA, May 23-26, 1988*, volume 310 of *Lecture Notes in Computer Science*, pages 162–181. Springer, 1988.

[14] G. Huet and J. Hullot. Proofs by induction in equational theories with constructors. *Journal of the Association for Computing Machinery*, 25(2), 1982.

[15] J.-P. Jouannaud and E. Kounalis. Proof by induction in equational theories without constructors. *Information and Computation*, 82(1), 1989.

[16] R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptology*, 7:79–130, 1994.

[17] D. Knuth and P. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational problems in abstract algebra*, pages 263–297. Pergamon Press, 1970.

[18] E. Kounalis and M. Rusinowitch. A mechanization of inductive reasoning. In AAAI Press and MIT Press, editors, *Proceedings of the American Association for Artificial Intelligence Conference, Boston*, pages 240–245, July 1990.

[19] G. Lowe. Breaking and fixing the Needham Schroeder public-key protocol using FDR. In *Proceedings of TACAS*, volume 1055, pages 147–166. Springer Verlag, 1996.

[20] D. Musser. On proving inductive properties of abstract data types. In *Proceedings 7th ACM Symp. on Principles of Programming Languages*, pages 154–162. ACM, 1980.

[21] L.C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6:85–128, 1998.

[22] M. Protzen. Disproving conjectures. In D. Kapur, editor, *11th Conference on Automated Deduction*, pages 340–354, Saratoga Springs, NY, USA, June 1992. Published as Springer Lecture Notes in Artificial Intelligence, No 607.

[23] W. Reif. The KIV Approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages and Tools for the Construction of Correct Software*, volume 1009. Springer Verlag, 1995.

[24] W. Reif, G. Schellhorn, and A. Thums. Fehlersuche in formalen Spezifikationen. Technical Report 2000-06, Fakultät fur Informatik, Universität Ulm, Germany, May 2000. (In German).

[25] W. Reif, G. Schellhorn, and A. Thums. Flaw detection in formal specifications. In *IJCAR'01*, pages 642–657, 2001.

[26] C. Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In H. Ganzinger, editor, *Automated Deduction – CADE-16, 16th International Conference on Automated Deduction*, LNAI 1632, pages 314–328, Trento, Italy, July 1999. Springer-Verlag.

Session:
Specification and Verification

Eager Formal Methods for Security Management

Amy L. Herzog Joshua D. Guttman

The MITRE Corporation
{aherzog, guttman}@mitre.org

June 28, 2002

Abstract

Achieving a security goal in a networked system requires the cooperation of a variety of devices, each potentially requiring different configurations. Many information security problems may be solved given appropriate models of these devices and their interactions. We present a unique approach, *eager formal methods*, which front-loads the contribution of formal methods to problem-solving.

With eager formal methods, we formally model the network and a class of practically important security goals. The models derived suggest algorithms which, given system configuration information, return security goals satisfied in the system. The formal methods provide rigorous justification, yet the algorithms are implemented as ordinary computer programs.

We have applied this approach to several problems; in this extended abstract we briefly describe two: distributed packet filtering and the use of IP security (IPsec) gateways. We have developed and implemented solutions to these two problems separately. We also describe how to unify the two solutions, jointly enforcing these mechanisms on a single network.

1 Introduction

Controlling complexity is a core problem in information security. Achieving a security goal in a networked system requires the cooperation of many devices, such as routers, firewalls, virtual private network gateways, and individual host operating systems. Different devices may require different configurations, depending on their purposes and network locations. Many information security problems may be solved given models of these devices and their interactions. We have focused for several years on these problems, using *eager formal methods* as our approach.

Eager formal methods front-loads the contribution of formal methods to problem-solving. The focus is on modeling devices, their behavior as a function of configurations, and the consequences of their interactions. A class of practically important security goals must also be expressible in terms of these models.

These models suggest algorithms taking as input information about system configuration, and returning the security goals satisfied in that system. In some cases, we can also derive algorithms to generate configurations to satisfy given security goals. The formal models provide a rigorous justification of soundness. By contrast, algorithms are implemented as ordinary computer programs requiring no logical expertise to use. Resolving practical problems then requires little time, and no formal methods specialists.

We have applied this approach to several problems. In this extended abstract, we briefly describe two problems and the modeling frameworks that lead to solutions. The first is the

distributed packet filtering problem, in which packet-filtering routers are located at various points in a network with complex topology. The problem is to constrain the flow of different types of packets through the network. The second problem concerns configuring gateways for the IP security protocols (IPsec); the problem is to ensure that authentication and confidentiality goals are achieved for specific types of packets traversing the network. Solutions to these problems have been published [3, 4, 5] and implemented. We also describe how to unify the two solutions, so that packet filtering goals and IPsec authentication and confidentiality are jointly enforced on a network.

Steps in Eager Formal Methods Our eager formal methods requires four steps, at least conceptually. In carrying out an eager formal methods project, "earlier" steps must often be revised so that later steps can succeed. The steps are:

Modeling Construct a simple formal model of the problem domain. For instance, in modeling packet filtering, the model contains a bipartite graph, in which nodes are either routers or network regions. Edges represent interfaces, and each interface may have packet filters representing the set of packets permitted to traverse that edge in each direction. Selecting this model will constrain what security goals are achievable or even expressible. Thus, simplicity in the model must be balanced against representing the right problems. The benefit of the model is to allow these problems to be solved systematically, in isolation from other issues raised by the reality of the system.

Security Goals Each model limits the properties of systems that are expressible. Within each model, we identify one or a few "logical forms" that define security-critical aspects of the systems. For instance, in our treatment of IPsec, one logical form characterizes assertions about authenticity; confidentiality is expressed using a different logical form. A particular system's security policy is expressed as a set of statements taking these logical forms.

Goal Enforcement The security goals and underlying model must be chosen so that there is an algorithm that, given a system as represented in the model, and a particular goal statement of one of the selected logical forms, determines whether the system satisfies that goal. Typically, different logical forms of security goals require different algorithms. In some cases, given some information about a system and some desired goal statements, an algorithm may fill in the details to describe a system satisfying the goals.

The essence of eager formal methods is to provide a rigorous proof of correctness for these algorithms.

Implementation Having defined and verified one or several goal enforcement algorithms, one writes a program to check goal enforcement. The inputs to this program consist of goal statements that should be enforced, and system configuration information. For instance, in our work, the system configuration information consists of network topology information, and the router configuration files. The program then enumerates which goals are met, and gives counterexamples for unmet goals. The algorithm may also generate new and better configuration files.

We will proceed now to illustrate this method in three successive cases.

2 Packet-Filtering Routers

Packet filtering routers are frequently an important component of network layer access control. The largest difficulty in implementing access control policies is that of localization. Since several

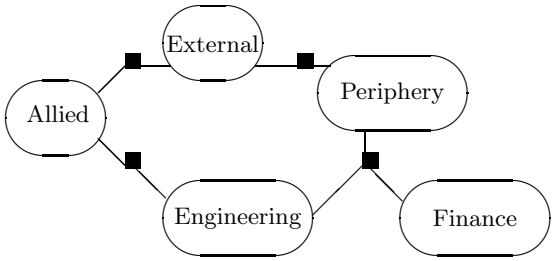


Figure 1: A Simple Example

different networks (and even companies) can be involved in an access control policy, routers in several locations sometimes must cooperate. It is difficult to manually determine what division of labor among the routers ensures policy enforcement, particularly since routing tables can be unpredictable. We will briefly introduce the network model, security goal format used, and discuss enforcement of this work before describing the tool suite implemented. Readers interested in a more formal treatment are directed to [3, 4].

2.1 Network Model

The network is regarded as a bipartite graph where the two types of nodes are *areas* and *routers*. Areas represent networks we wish to separate; routers connect the areas and move packets between them. There is an edge between a router and an area if the router has an interface on that area. (In Figure 1, *Engineering*, *External*, and *Allied* are all areas, and the black squares indicate routers.) A path through the network is a sequence of immediately connected nodes on the associated bipartite graph. This framework allows us to ignore issues of routing, so that our conclusions will hold even on the (realistic) assumption that routing tables change. Note that within this framework, access control security goals can only be enforced if they involve flow of packets from one area to another. Thus, the security goals themselves determine the granularity of the model.

Sets of packets form a boolean algebra, and the behavior of a system consists of the boolean algebra of sets of packets traversing the edges of a bipartite graph. The edges of the graph represent the interfaces to routers. Each router has a filter to apply to packets traversing the edge inbound into the router, and another filter to apply to packets traversing the edge outbound from the router. These filters are also regarded as sets of packets. Thus, the packets that survive all of the filters along a given path may be calculated if we are given the boolean algebra of sets of packets. Likewise, the set of packets that can pass from one given area to another (along any path in the graph) may be calculated using a graph algorithm that calls boolean algebra operations as a primitive. Other algorithms may be used to calculate what filters should be attached to specific edges so that the network will satisfy useful constraints.

We implement the boolean algebras using binary decision diagrams [2, 1], a representation that is reasonably efficient for the sets that arise naturally in actual networks [4].

2.2 Security Goals

These security goals rely upon two sorts of information: areas the packet has actually traversed, and packet header contents. The goal statements concern two distinct areas occurring in the actual path of the packet, and a predicate ϕ of packets (such as membership in a certain set of abstract apackets). If p ranges over packets, then

If p was previously in a_1 and later reaches a_2, then $\phi(p)$.

is a *policy statement* when $a_1 \neq a_2$. It requires that a_2 be protected against non-ϕ packets if they have ever been in a_1. For example:

> If p was ever in the `External` area and later reaches the `Engineering` area, then p should have service `smtp`.

Observe that the security goals are designed to be meaningful properties of a boolean algebra of sets of packets traversing a bipartite graph.

2.3 Goal Enforcement

Goals are enforced by assigning filters to routers. Filters, as mentioned above, are sets of packets, representing all packets allowed to pass that filtering point.

To check that a set of filters enforces a given security policy, we examine each path between the appropriate areas to ensure that the set of packets that survive all filters traversed is included in the policy statement. The set of packets which survives all filters on a path is called the *feasibility set* for that path. For a path σ from a_1 to a_2, we check that the feasibility set for σ is included in the set of abstract packets permitted (according to policy) to travel from a_1 to a_2.

One can use a variant of this checking algorithm to generate a filtering assignment as well: start with an extremely permissive filtering assignment, and check the assignment against a desired policy to see where the assignment fails. One can then "tighten" the filter assignments until the policy is enforced.

2.4 Implementation: The Network Policy Enforcer

This method for checking a set of routing filters against a policy was implemented in 1997 in the Network Policy Tool (NPT) [3]. Since that time, NPT has been updated, and packaged with a suite of complementary tools to form the Network Policy Enforcer (NPE). NPE is concerned with the entire life-span of policy discovery, analysis, and enforcement. It begins by building a network map from router, switch, and firewall configuration files. It parses vendor-specific configuration files into Binary Decision Diagrams representing the sets of packets that survive each filter.[1] It calculates from these filters an "effective policy" containing the most permissive policy statements enforced by the given collection of configuration files. It also calculates the violations of this effective policy relative to an administrator-defined desired policy, suggesting local tightening for specific filters. NPE also records the changes in the effective policy over time, as the network evolves.

3 The IP Security Protocols (IPsec)

The IP security protocols (see [9, 7, 8], and also [10, 6]), collectively termed IPsec, are an important set of security protocols that include ensuring confidentiality, integrity, and authentication of data communications in an IP network. The IPsec protocols are a set of recipes for applying cryptographic transformations to packets traversing the network. The transformations include encryption for confidentiality and computing a keyed hash for authentication. A new header is prefixed to the packet when these operations occur. This header is removed when the peer does a matching decryption or checks the keyed hash. The peers performing the cryptographic

[1] Currently, only the Cisco configuration language is supported. Other languages are, however, not too different.

operations may be the original source and final destination of the packet, or they may also be security gateways operating on their behalf.

In [5], we formalized the types of security goal that IPsec is capable of achieving. Continuing our use of eager formal methods, we then provided criteria that entail that a particular network achieves its IPsec security goals. We present a brief overview of past work; the interested reader is directed to [5] for more information.

3.1 Network Model

We again regard the network as a bipartite graph, containing *areas* as before, and *devices*, representing the hosts or gateways which perform IPsec processing. A location l is a node of such a network graph. We must extend the previous network model specifically in the area of headers, which IPsec layers and modifies.

Formally, a *header* is a triple consisting of a source, a destination, and a protocol data value. This protocol data value may mark the packet as `ftp`, `telnet`, or `sunrpc` using the `tcp` or `udp` port information, or it may reflect the fact that the packet has a cryptographic IPsec header. If the set of all protocol data is P, then $A \subset P$ is a set of *authenticated* protocol data (and represents those headers that provide IPsec authentication and integrity services). The set $C \subset A$ is a set of *confidentiality* protocol data. To reason about headers, we pay strict attention to the *state* of a packet.

Definition 1 *A packet state is a non-empty sequence $\langle h_1, \ldots, h_n \rangle$ of headers h_i.*

Since header sequences may be arbitrarily long, we require some way of determining if any more IPsec processing should happen at a given location. The special processing state `ready`, denoted κ, indicates that the packet is now ready to move through an interface to a device or network.

The travels of a packet through a system are modeled as the evolution of a state machine. The packet may not yet have started to travel; this is the start state. The packet may no longer be traveling; this is the finished state. Every other state is a triple of a node l in the graph, indicating where the packet is currently situated; a processing state indicating if the packet is ready to move; and a packet state, indicating the sequence of headers nested around the payload of the packet.

State machine transitions include packet creation, discard, and movement operators; header prefixing and popping operators; and null operators (which change only the processing state). The total set of transitions is constrained both by security goals and the actual network graph–for example, the `move` operator is undefined when there is no edge between the two relevant nodes.

The state machine corresponding to a particular network configuration is determined using the same modeling ideas as before. The bipartite graph determines the possible atomic motions of the packet, which can only move across from a node to an adjacent edge (or vice versa). The IPsec configuration indicates which cryptographic headers must be added or removed at each processing location. At each processing location, there are sets of packets for which a type of header will be added, as well as sets of packets that are permissible results when a cryptographic header is removed. The additional state machine structure in analyzing IPsec is needed because the processing may change the form of the packet.

3.2 Security Goals

IPsec can achieve the two primary security goals of authentication and confidentiality. Authentication allows a packet recipient to take a packet "at face value". Thus, for a packet p selected for protection by an authentication goal,

> If A is the value in the source header field of p as received by B, then p actually originated at A in the past, and the payload has not been altered since.

Confidentiality goals provide for the privacy of packet content. The confidentiality goal for a packet with source field A, requiring protection from disclosure in some network location C, stipulates:

> If a packet originates at A, and later reaches the location C, then while it is at C it has a header providing confidentiality.

(The proviso that the packet was once at A is necessary, because in most cases we cannot prevent someone at C from creating a spoofed packet. However, a spoofed packet cannot compromise A's data if it has no causal connection to A.)

3.3 Goal Enforcement

We make the assumption that cryptographic operations work as advertised, and hence that any tampering with a cryptographically protected packet can be detected. This allows us to view cryptographic operations as a tunnel; while cryptographic protection is in place any tampering with the packet will cause the packet to be dropped. Thus the set of locations accessible before or after cryptographic tunnel protection are the only ones of real importance for goal achievement. We will call these locations outside tunnel protection *trust sets* for particular security goals. A trust set is goal-specific, and not typically connected. Of special importance are those systems inside a trust set with a direct connection to outside systems: the trust set *boundary*.

Goal enforcement is achieved via behavior restrictions on trust set members, dependent on both the goal and if the member is also in a boundary.

Authentication. The *Creation Rule* states that trust set nodes must not spoof packets with trust set sources. The *Pop Rule* codifies the notion that only nodes in the trust set should be able to certify a packet as coming from the trust set. It states that if a device processes an IPsec packet p with source not in the trust set, and removing this header leads to a packet p' with source *in* the trust set, p' must be discarded. Finally, the *Inbound Ready Rule* requires that the boundaries of the trust set must not pass an inbound packet which should have presented authentication headers, but did not.

Confidentiality. Two behavior restrictions apply to trust set members for confidentiality goals. The *Destination Prefix Rule* ensures that whenever an IPsec system inside the trust set adds a confidentiality header to a packet requiring protection, the source and destination of the added header are also in the trust set. Boundary IPsec devices must also abide by the *Outbound Ready Rule*: the device passes a packet p only if its topmost layer is a confidentiality header, or else it has no IPsec headers and p does not require confidentiality.

These purely local behavior restrictions guarantee that authentication and confidentiality goals will be met [5].

3.4 Implementation: The Confidentiality and Authentication IPsec Checker (CAIC)

Checks for these behavior restrictions were implemented in the "Confidentiality and Authentication IPsec Checker" (or CAIC, pronounced "cake"). When given Cisco IPsec configuration files and a network / policy specification, CAIC will check trust sets and boundaries for the behavior restrictions described above. It returns to the user both a verdict (the goal is enforced

or not) and a description of goal failure (if appropriate). It describes which behavior restriction was not met, and the specific sorts of packets upon which the goal fails. CAIC shares most of its implementation with NPE, and uses Binary Decision Diagrams to represent the packet sets relevant to each network configuration.

4 Combined Packet Filtering and IPsec

In protecting its communication, an organization is likely to use a variety of different tools in tandem. Of special interest here, most companies use both packet-filtering firewalls and the IPsec protocols (for example, as part of a VPN). We have presented mechanisms that ensure that both sorts of security goals are achievable individually, but the use of these two tools in a heterogeneous environment presents complications.

Recall that packet headers were assumed to be unchanged when network access control was the only aim. IPsec is primarily concerned with the addition and deletion of packet headers; thus when the two tools are used in conjunction the assumptions underlying the packet filtering firewall analysis are false, and the use of IPsec could easily compromise the achievement of packet-filtering goals.

We combine the approaches of [5, 3] retaining the eager formal methods approach, and emerge with a more general framework, capable of ensuring goal achievement for both types of objective.

4.1 Network Model Modifications

In addition to nodes representing network and devices, we add nodes corresponding to interfaces on devices, using the *Enriched System Representation* described in [5]. This allows us to represent the conceptual location of a packet when either IPsec or packet-filtering processing is occurring, as the packet is traversing either the inbound or the outbound interface to (or from) that device. We call these conceptual locations *directed interfaces*. This produces a directed graph with three kinds of nodes–networks, devices, and directed interfaces. A location l is a node, and packet states are as before: non-empty sequences $\langle h_1, \ldots, h_n \rangle$ of packet headers.

To simplify our reasoning as much as possible, we will be very concerned with the *originator* of a given packet; we will see the use of this notion in Section 4.2.

One remaining network modification relates to the notion of a *trust set* (required for IPsec-style security goals). This notion of systems "near" the source and destination of a particular packet holds significance even for non-IPsec packets. Rather than denoting potentially vulnerable locations, they represent the "last line of defense" for any filtering to occur.

Definition 2 *A **pre-filtering set** (respectively, a **post-filtering set**) S for $G = (V, E)$ consists of a set $R \subset V$ of networks, including the source node (respectively, the destination node) of goal-relevant packets, together with all devices g adjacent to networks in R and all interfaces $i_g[*]$ and $o_g[*]$.*

The inbound boundary of S, written $\partial^{in} S$ is the set of all interfaces $i_g[r]$ or $i_g[g']$ where $g \in S$ and $r, g' \notin S$.

The outbound boundary of S, written $\partial^{out} S$ is the set of all interfaces $i_g[r]$ or $i_g[g']$ where $g \in S$ and $r, g' \notin S$.

A location ℓ is responsible *for a header state h if either h is of length 1 and ℓ originated the packet, or ℓ is of length greater than 1 and ℓ affixed the outermost header in the sequence h.*

Pre-filtering sets are those locations close to the source of a packet, and represent the first places filtering can occur. Post-filtering sets are those locations close to the destination of a packet, and represent the last places filtering can occur.

The remainder of the network specification is as described in [5].

4.2 Uniform Goal Statements

In order to reason about the achievement of goals in a heterogenous network, we require a goal format which represents the three types of security goals uniformly. Our security goals will be a collection of one or more statements (formulae involving the trajectory of the packet). Focusing on two locations n and n', there will be a constraint ϕ on header states while at n and n', based on a set of headers of interest s at some early location. This allows the statements to focus on simple headers (the packet header while at the first node) rather than some later node with a potentially complex and arbitrary header stack. A general statement follows here:

if n responsible for h and $h \in s$
and later state (n', κ, h')
then $\phi(h, h')$

It is straightforward to represent all security goals in this format. We describe only confidentiality goals here.

Confidentiality Goals. Confidentiality goals should protect packets from ever appearing in "dangerous" areas without a confidentiality header. A general form of the goal:

if ℓ_1 responsible for h and $h \in s$
and later state (ℓ_2, κ, h')
then $h' = \langle \cdots (x, y, \texttt{cnf}) \cdots h \rangle$

We briefly mention goal equivalence before addressing enforcement.

4.3 Goal Achievement

Given that the general security goal statements are equivalent to those in [3, 5], previous proofs for goal achievement are still valid when tools are used alone. In a heterogeneous environment, IPsec security goals are still provably achievable–any packet filtering goal could not affect IPsec filtering. headers directly; they can only cause packets to be passed or with service requirements, packet goal achievement.

Authentication and confidentiality goals can affect the achievement of packet-filtering goals, however. The addition of headers may obscure the payload of a packet, perhaps causing the packet to miss some crucial filtering. A trivially provable solution to this problem is simply to install a complete set of filters on every router in the system. This solution is certainly sub-optimal. We therefore offer a description of filter assignment which is more efficient, and ensures that filtering is never missed, regardless of IPsec activity.

4.3.1 Filter Assignment

Before any filter adjustment, IPsec security goal enforcement rules are put in place; this involves the choosing of trust sets for the goals. (We discuss the tensions related to choosing trust sets in Section 4.4.) For a specific packet-filtering goal, the pre- and post-filtering sets are then chosen. To ensure achievement of this goal, we impose two behavior restrictions in the style of [5] on pre- and post-filtering set members. Fix a pre- or post-filtering set S.

Inbound Protection Rule (Part One). For any transition

$$(\ell_1, \kappa, h)$$
$$\Downarrow$$
$$(\ell_2, \kappa', h)$$

If $\ell_2 \in \partial^{in} S$ and $\ell_1 \in S$, then ℓ_2 discards the packet. This rule ensures boundary members discard externally forged packets.

Given the presence of IPsec, we also must impose an IPsec restriction on all *pre- and post-filtering set* members. Even in the absence of an authentication goal, we do not wish to grant devices outside the pre- and post-filtering set the authority to authenticate systems inside.

Inbound Protection Rule (Part Two). For any transition

$$(\ell, \kappa, \langle [s, d, A], [a, -, -] \rangle) \Downarrow (\ell, \kappa', \langle [a, -, -] \rangle)$$

If $\ell \in S$ and $a \in S$, then $s \in S$.

If these behavior restrictions hold, then provided the filtering rules are applied, goal achievement is ensured. To assign filtering rules, first use the method described in [3]. Then use the following procedure, expanding and contracting the set of filters in the system.

For any filtering at any point within a pre- or post-filtering set, duplicate the filters to all other tunnel entrance/exit points in the set. This ensures that regardless of where in a pre- and post-filtering set a packet could enter or exit a tunnel, filtering occurs.

Then pick a filtering point p not in either the pre- or post-filtering set. Assume it implements a filter ϕ. Trace all non-cyclic paths from p to the destination portion of the pre- and post-filtering set. At each potential tunnel endpoint along each path, assign ϕ. Repeat this process until all filtering points outside the post-filtering set have been processed. Repeat this filter expansion for every filtering point outside the pre- and post-filtering sets.

To contract unnecessary filters, now trace all non-cyclic paths between the source and destination of the filtering goal. If some individual filtering rule ψ happens on every path regardless of tunnel activity, it can be removed from all filters on the paths previous to the eventual occurrence.

The filters are now assigned and will enforce the packet-filtering security goal if the following heuristic is used when a packet emerges from a tunnel.

When To Discard. Upon exiting a tunnel, a packet must receive any missed filtering. There are two types of tunnel in this case–tunnels with a source in the pre- or post-filtering set (which authenticate senders in the set), and ad-hoc tunnels which may meet user-defined goals.

When a packet p emerges from a tunnel of the first type, goals are checked for $p.1.src$ and $p.1.dst$. If $p \in$ `Safe`, the packet is passed; otherwise, discarded. When a packet p emerges from a tunnel of the second type, $p.1.src$ is not authenticated, so goals must be checked for all origins and $p.1.dst$. p is discarded if there exists a source such that p is not a member of `Safe`.

4.3.2 Why This Works

It is clear that IPsec goals are guaranteed achievement if the behavior restrictions laid out in [5] are met, regardless of packet-filtering activity. In the absence of any IPsec tunnels, packet-filtering goal achievement is ensured as outlined in [3].

The question, then, is of filtering goals in the presence of IPsec tunnels. There are four cases:

Both tunnel endpoints in pre- / post-filtering set. Forged packets cannot enter the pre- and post-filtering sets due to the Inbound Protection Rules, and the tunnel exit-point can verify (via cryptographic means) the veracity of the tunnel, and then trust the tunnel source to authenticate the packet source.

Only tunnel entrance in pre- / post-filtering set. Achievement of filtering goals relies upon our filtering assignment. If the packet would normally have missed filtering, our assignment ensures that the necessary filters are along all paths to the destination.

Only tunnel exit in pre- / post-filtering set. When the packet exits the tunnel, the exit-point will not trust any authentication present for the source, and will check goals with the destination of the packet for *all* sources. This will prevent a forged packet from successfully being delivered, and hence ensure goal enforcement.

Neither tunnel endpoint in pre- / post-filtering set. In this case, we can be assured through our filter assignments that any necessary filtering will happen.

Thus regardless of tunnel activity, if the filtering assignments and behavior restrictions above are followed, packet-filtering goals are achieved. Management of goals in this heterogeneous environment can be cumbersome. We briefly present strategies to minimize that burden.

4.4 Management Tactics

It is possible to construct networks and goals such that in order that packet-filtering goals be achieved, nearly every device in the system must have a full copy of all filters. Given some care in design, this is very unlikely.

Most filtering problems arise when relatively short IPsec tunnels cross some sort of "boundary". There are two ways in which this can happen. First, the "boundary" crossed may be literal: A tunnel may begin in a pre- or post-filtering set, and end outside the set (or vice-versa). The second type of "boundary" is a filtering point. In traditional filter assignment a filter ϕ may be assigned to a device far from a pre- or post-filtering set. If a tunnel of one or two hops crosses that filtering point, a boundary has been crossed (this type of tunnel necessitated our filtering assignment "expansion").

There are two immediate tactics to reduce the annoyance of these short tunnels. The first tactic to reduce management burden is to ensure that pre- and post-filtering sets match existing IPsec trust sets. This is simpler and ensures that a higher percentage of legal packets will actually be delivered (since a greater number of packets exiting tunnels will have verifiable sources).

In addition to coordinating pre- and post-filtering sets with trust sets, one can also attempt to eliminate these short tunnels entirely. The first type of boundary hopping tunnel discussed above can be avoided by coordinating the filtering and trust sets as above. The second can be avoided in two ways: First, alter security goals so that they do not require filters outside of the pre- or post-filtering set (which may perhaps be a wise choice anyway; it is possible such devices may not be truly trustworthy). One could also limit the number of IPsec capable devices in the network to advantageous locations.

5 Conclusion

We have introduced eager formal methods, a verification method that differs in approach from more common methods: First, formally outline general security properties one desires from a given mechanism (such as packet-filtering). Then find a simple set of conditions which, if met, imply the security property is held. This implication is formally verified, allowing for quick checks of simple conditions for each instantiation of the mechanism. We have seen that many security desires can be verified in this way; from authentication and confidentiality goals to standard packet filtering aims, to a combination.

Many advantages are granted by this approach. It is efficient, allowing for the time-consuming task of formal verification to be done only once. Further, this verification can be separate from

any property-checking tools, allowing those tools to be implemented quickly, and run efficiently. We illustrated this approach by reprising previous work on packet-filtering and IPsec formal verification. We also introduced a new instance of this verification approach which ensures achievement of both packet-filtering goals and IPsec desires.

In addition to this work, there are many future possibilities for eager formal methods. Policy analysis work for Mandatory Access Controls is ongoing, and follows the same procedure.

References

[1] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45, 1990.

[2] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[3] Joshua D. Guttman. Filtering postures: Local enforcement for global policies. In *Proceedings, 1997 IEEE Symposium on Security and Privacy*, pages 120–29. IEEE Computer Society Press, May 1997.

[4] Joshua D. Guttman. Security goals: Packet trajectories and strand spaces. In Roberto Gorrieri and Riccardo Focardi, editors, *Foundations of Security Analysis and Design*, volume 2171 of *LNCS*. Springer Verlag, 2001.

[5] Joshua D. Guttman, Amy L. Herzog, and F. Javier Thayer. Authentication and confidentiality via IPsec. Technical report, The MITRE Corporation, March 2000.

[6] D. Harkins and D. Carrel. *The Internet Key Exchange (IKE)*. IETF Network Working Group RFC 2409, November 1998.

[7] S. Kent and R. Atkinson. *IP Authentication Header*. IETF Network Working Group RFC 2402, November 1998.

[8] S. Kent and R. Atkinson. *IP Encapsulating Security Payload*. IETF Network Working Group RFC 2406, November 1998.

[9] S. Kent and R. Atkinson. *Security Architecture for the Internet Protocol*. IETF Network Working Group RFC 2401, November 1998.

[10] D. Maughan, M. Schertler, M. Schneider, and J. Turner. *Internet Security Association and Key Management Protocol (ISAKMP)*. IETF Network Working Group RFC 2408, November 1998.

High-performance deduction for verification: a case study in the theory of arrays

Alessandro Armando
DIST
Università degli Studi di Genova, Italy
armando@dist.unige.it

Maria Paola Bonacina *
Aditya Kumar Sehgal†
Department of Computer Science
The University of Iowa, USA
{*bonacina, asehgal*}*@cs.uiowa.edu*

Silvio Ranise‡
Michaël Rusinowitch
LORIA & INRIA-Lorraine, Villers-lès-Nancy, France
{*Silvio.Ranise, Michael.Rusinowitch*}*@loria.fr*

Abstract

We outline an approach to use ordering-based theorem-proving strategies as satisfiability procedures for certain decidable theories. We report on experiments with synthetic benchmarks in the theory of arrays with extensionality, showing that a theorem prover – the E system – compares favorably with the state-of-the-art validity checker CVC.

1 Introduction

Satisfiability procedures for theories of standard data-types, such as arrays, lists, bit-vectors, are at the core of most state-of-the-art verification tools (e.g., ACL2 [8], PVS [12], Simplify [7], CVC [18]). They are required for a wide range of verification tasks and are fundamental for efficiency. Satisfiability problems have the form $T \cup S$, where S is a set of ground literals (read as conjunction), T is a *background theory*, and the goal is to prove that $T \cup S$ is unsatisfiable.

The endeavour of designing, proving correct, and implementing a satisfiability procedure for each decidable theory of interest is far from simple. First, most problems involve more than one theory, so that one needs to *combine satisfiability procedures* [11, 16]. Combination is complicated: for example, understanding, formalizing and proving correct the method in [16] required significant effort (e.g., [14]). With a theorem prover, one may simply give in input the union of the axiomatizations of the theories. Second, every satisfiability procedure needs to be proved correct and complete: a key ingredient is to show that whenever the algorithm reports "satisfiable," its

*Supported in part by NSF grant CCR-97-01508 and a Dean Scholar Award, The University of Iowa.
†Supported in part by NSF grant CCR-97-01508.
‡Also supported by Università degli Studi di Genova.

ouput represents a model of $T \cup S$. Model-construction arguments can be complex, and the more concrete is the description of the procedure, the more difficult are the proofs (e.g., [14, 19]). If one develops an abstract framework (e.g., [2]), the additional clarity is gained at the expense of proximity with concrete procedures. On the other hand, if a theorem-proving strategy has a sound and refutationally complete inference system, and a fair search plan, it is a semi-decision procedure for unsatisfiability, and we can use it without additional proofs.

Given these attractive features of theorem proving, it would be all the more precious if we could exclude the risk of non-termination on the decidable theories of interest. Results of this nature were presented recently in [1], for the theories of *lists*, *arrays with extensionality*, and their combination, among others. The analysis in [1] showed that a standard, paramodulation-based inference system, for first-order logic with equality, is guaranteed to terminate on $T \cup S$, if T is any of the above theories. The proofs of termination rest on case analyses demonstrating that the inference system can generate only finitely many clauses from such inputs. Thus, a strategy that combines this inference system with a fair search plan is *in itself* a decision procedure for satisfiability in those theories, and a theorem prover that implements it can be used *off the shelf* as a validity checker. One could question whether a specific theorem prover is a sound and complete implementation of such a theorem-proving strategy. However, this question applies also to a validity checker implementing decision procedures, and perhaps even more seriously, considering the common practice of designing and implementing from scratch both data structures and algorithms for each new procedure. In contrast, a deduction-based approach also has potential for better software reuse, since one can envision constructing satisfiability procedures, by combining the generic reasoning modules offered by state-of-the-art theorem provers.

Even after termination has been proved and a higher degree of assurance about the soundness of the procedure can be offered, the issue of efficiency remains. The general expectation is that an implementation of a satisfiability procedure, with the theory built-in as a background theory, will be always much faster than a theorem prover that takes T in input. In this paper, we suggest that this may not be obvious. We consider *synthetic benchmarks*, because they allow to assess the scalability of an approach by experimental asymptotic analysis. We propose two sets of synthetic benchmarks in the *theory of arrays with extensionality*, and we report on experiments with two tools: the E theorem prover [15], and the CVC validity checker [18]. E implements (a variant of) the inference system used in [1] with several search plans. CVC combines decision procedures in the style of [11], as described in [3], including that of [19] for the theory of arrays with extensionality, and featuring either GRASP [17] or Chaff [10] as propositional solver. The experiments show that, for both sets of benchmarks, there is a configuration of the general-purpose prover that is competitive with the validity checker. This is preliminary, encouraging evidence that the approach of [1], in addition to being theoretically elegant, is also applicable in practice.

2 A deduction-based approach and the E prover

The termination results of [1] require that the literals in S be *flat*. This means that the sum of the depths of the sides of an equation, or disequation, has depth at most 1, or at most 0, respectively (assuming constants and variables have depth 0). Literals that are not flat can be flattened by

introducing new constant symbols:

Example 1 *Assume the function symbols store*: ARRAY × INDEX × ELEM*ent* ⟶ ARRAY *and select*: ARRAY × INDEX ⟶ ELEM*ent denote the operations of storing and retrieving a value at a position in an array, respectively (e.g., [11]). The ground literals in* $S = \{store(s, a, v) = store(s_1, a_1, v_1);\ select(s, a) = v;\ select(s_1, a_1) = v_1;\ a = a_1;\ v \neq v_1\}$ *can be flattened by introducing new constants* c_1, c_2, c_3, c_4*, yielding* $S' = \{store(s, a, v) = c_1;\ store(s_1, a_1, v_1) = c_2;\ select(s, a) = c_3;\ select(s_1, a_1) = c_4;\ c_1 = c_2;\ c_3 = v;\ c_4 = v_1;\ a = a_1;\ v \neq v_1\}$*. This trasformation preserves satisfiability:* $T \cup S$ *is satisfiable if and only if* $T \cup S'$ *is, for arbitrary* T*.*

Flattening can be done in different ways: we call it *strict*, if all occurrences of a subterm are replaced by the same constant, and *non-strict*, if each subterm occurrence is replaced by a new constant. Non-strict flattening yields an under-constrained problem, whose unsatisfiability obviously implies unsatisfiability of the strictly flattened version. Strict flattening minimizes the number of new constants introduced, hence the number of clauses, "sharing" subterms as much as possible. A non-strictly flattened version has less sharing of subterms and more clauses. After this pre-processing, $T \cup S$ can be given to any fair theorem-proving strategy with the following inference system (e.g., [1]): *ordered superposition/paramodulation*, *reflection* (also known as *equality factoring*), and *ordered factoring*, as expansion inference rules, and *subsumption*, *simplification* and *deletion* (of trivial equations), as contraction inference rules.

Like most ordering-based provers, E implements search plans based on the *given-clause loop* [9]. The prover works with two lists of clauses, say *To-be-selected* and *Already-selected*: at every iteration, it extracts a clause, the *given clause*, from *To-be-selected*, moves it to *Already-selected*, performs all expansion inferences between the given clause and clauses in *Already-selected*, and appends the normal forms of all new clauses thus generated to *To-be-selected*. Practical ordering-based strategies require that contraction be applied *eagerly*, to avoid generating clauses from clauses that can be deleted by contraction. Variants of the given-clause loop differ in the implementation of eager contraction: while the Otter version aims at keeping the union of *To-be-selected* and *Already-selected* inter-reduced, E implements a version that keeps only *Already-selected* inter-reduced, on the ground that all parents of expansion inferences are in *Already-selected*, with the downside that clauses in *To-be-selected* are not applied as simplifiers.

The features of the search plans in E that were most relevant to our experiments are *clause selection* and *term ordering*, and *literal selection* to a lesser extent. For clause selection, given a heuristic evaluation function f, the given-clause loop implements a *best-first search*, by selecting at each iteration a clause C such that $f(C)$ is minimum. E uses pairs of functions (f_1, f_2), where f_1 is the *clause priority function* and f_2 the *heuristic weight function*, to pick a clause of smallest weight among those of highest priority. Ties are broken by selecting the oldest clause. Every pair (f_1, f_2) defines a priority queue: E allows the user to activate and weight any number of them, resulting in a weighted round robin scheme.

Considering that our problems have the form $T \cup S$, one may think of a *set-of-support strategy*, with T as consistent set and S as set of support. However, E does not emphasize supported strategies, because using a set of support is complete for resolution, but not for (ordered) resolution and paramodulation, unless T is saturated. If T is saturated, by definition, all inferences from T

are redundant, and therefore using its complement as set of support does not add focus to the search: e.g., T may generate very few clauses that are subsumed right away. This is the case for the first presentation considered in our experiments (named T_1 and introduced in Section 3): its two axioms generate only one trivial clause. The second presentation we used (named T_2 and also introduced in Section 3), is not saturated, so that one could consider using T_2 as consistent set and S as set of support, since incomplete strategies are often used in experiments. However, E, unlike Otter, does not let the experimenter choose the input set of support: with its clause priority function `SimulateSOS`, which prefers supported clauses, the set of support is initialized by the prover to contain the input negative clauses. Nevertheless, we chose to use it.

Among weight functions, we tried both `Clauseweight` and `Refinedweight`. The former, e.g., `Clauseweight(x,y,z)`, is the number of symbols in the clause, with weights `x` for non-variable symbols and `y` for variable symbols, and the resulting weight of each positive literal multiplied by `z`. `Refinedweight(x,y,z,w,t)` is similar, but aims at taking the term ordering into account, by multiplying the resulting weight of each maximal term and maximal (or selected) literal by `w` and `t`, respectively. The *term ordering* is the ordering on terms and literals used for well-founded rewriting and to restrict paramodulation/superposition. E implements *Knuth-Bendix ordering* (KBO), and *lexicographic path ordering* (LPO) (e.g., [6]), and we experimented with both. These orderings require a *precedence* on function symbols that can be given by the user, built by the prover, or a combination of the two. KBO also requires to weight the symbols: by default, E assigns all symbols weight 1, except the first non-constant maximal symbol which gets weight 0. The *literal selection* functions select literals in clauses to restrict ordered paramodulation further. We tried a few and settled for `SelectComplex`: it selects the first literal in the form $x \neq y$; if the clause has none, it picks the smallest ground negative literal; if the clause has none, it picks an arbitrary negative literal among those with the largest difference in number of symbols between left and right side. In *automatic mode*, E determines automatically clause evaluation function, term ordering and literal selection function for the given input.

3 Synthetic benchmarks in the theory of arrays

The presentation of the theory of arrays with extensionality is given by the following axioms:

$$\forall A, I, E.\ select(store(A, I, E), I) = E \tag{1}$$

$$\forall A, I, J, E.\ (I \neq J \Longrightarrow select(store(A, I, E), J) = select(A, J)) \tag{2}$$

$$\forall A, B.\ (\forall I. select(A, I) = select(B, I) \Longrightarrow A = B) \tag{3}$$

where A and B are variables of sort ARRAY, I and J are variables of sort INDEX, and E is a variable of sort ELEMent. The clausal forms of axioms (1) and (2) are given in input to the prover together with the ground literals of the specific problem. Axiom (3), the *extensionality axiom*, is a universal-existential formula of sorted first-order logic with equality, which is not given to the prover, but handled by pre-processing the input set of ground literals [1]. This pre-processing step consists of replacing every disequality of the form $t \neq t'$, where t and t' are terms of sort ARRAY, by the disequality $select(t, sk(t, t')) \neq select(t', sk(t, t'))$, where sk is a skolem function of type ARRAY $\times$ ARRAY $\longrightarrow$ INDEX. Indeed, this is the result of applying a resolution step to $t \neq t'$ and

the clausal form of axiom (3), $select(A, sk(A,B)) \neq select(B, sk(A,B)) \vee A = B$. Unsatisfiability in the theory is clearly preserved. Intuitively, $sk(t,t')$ is an index where the arrays t and t' differ.

An alternative axiomatization of the theory of arrays with extensionality (e.g., [11]), leaves axioms (1) and (2) unchanged, and replaces (3) by:

$$\forall A, I.\ store(A, I, select(A, I)) = A \tag{4}$$

$$\forall A, I, E, F.\ store(store(A, I, E), I, F) = store(A, I, F) \tag{5}$$

$$\forall A, I, J, E.\ (I \neq J \Longrightarrow store(store(A, I, E), J, F) = store(store(A, J, F), I, E)) \tag{6}$$

where A is a variable of sort ARRAY, I and J are variables of sort INDEX, and E and F are variables of sort ELEMent. We shall refer to the first axiomatization as T_1 and to the second one as T_2. An axiomatization for *finite maps* similar to T_2 was given in [5] together with a model built in HOL: it includes axioms (1), (2), (5), (6), plus an induction principle that allows one to derive (3) and (4) as theorems. One can easily prove by hand that T_1 entails T_2, so that if $T_2 \cup S$ is unsatisfiable, $T_1 \cup S$ is unsatisfiable also. T_2 was not used in [1] and there is no termination result for this presentation. Not surprisingly, saturation of T_2, that we tried on the side of our experiments with E, did not terminate. Thus, when working with T_2, the theorem-proving strategy acts as a semi-decision procedure, taking in input the clausal form of $T_2 \cup S$.

We present two sets of synthetic benchmarks for this theory. For the first one, the idea is to express the "commutativity" of storing elements at distinct places in an array a. Let $\{k_1, \ldots, k_N\}$ be N indices and C_2^N denote the set of 2-combinations over $\{1, \ldots, N\}$. To say that they are distinct, we write $\bigwedge_{(p,q)\in C_2^N} k_p \neq k_q$: e.g., for $N = 3$, $k_1 \neq k_2 \wedge k_1 \neq k_3 \wedge k_2 \neq k_3$. Then, if $i_1, ..., i_N$ and $j_1, ..., j_N$ are two distinct permutations of $1, \ldots, N$, the equation $store(\ldots(store(a, k_{i_1}, e_{i_1}), \ldots k_{i_N}, e_{i_N})\ldots) = store(\ldots(store(a, k_{j_1}, e_{j_1}), \ldots k_{j_N}, e_{j_N})\ldots)$ captures the desired property. For example, for $N = 3$, and permutations $(1,2,3)$ and $(2,1,3)$, we get $store(store(store(a, k_1, e_1), k_2, e_2), k_3, e_3) = store(store(store(a, k_2, e_2), k_1, e_1), k_3, e_3)$. Altogether we have the following schema:

$$(\textstyle\bigwedge_{(p,q)\in C_2^N} k_p \neq k_q) \Rightarrow$$
$$store(\ldots(store(a, k_{i_1}, e_{i_1}), \ldots k_{i_N}, e_{i_N})\ldots) = store(\ldots(store(a, k_{j_1}, e_{j_1}), \ldots k_{j_N}, e_{j_N})\ldots).$$

Each choice of permutations generates a different instance of the schema, and since there are $N!$ permutations of $\{k_1, \ldots, k_N\}$, the number of instances is the number of 2-combinations of $N!$ permutations, hence $\binom{N!}{2}$ or $(N!(N!-1))/2$. In our experiments, for each value of N, we sampled at most 10 permutations, hence 45 instances, in order to reduce the dependence of the results on the structure of the formula. We use $storecomm(N)$ to denote the generated instances for size N and the problem of checking their validity, or the unsatisfiability of their negation.

For the second group, the intuition is that swapping pairs of elements in an array a in two different orders yields the same array. The equations can be defined recursively. In the base case, $p = 0$, $k = 2p = 0$, and for $N = k + 2 = 2$ elements, the equation is $L_2 = R_2$, where

$$L_2 = store(store(a, i_1, select(a, i_0)), i_0, select(a, i_1))$$
$$R_2 = store(store(a, i_0, select(a, i_1)), i_1, select(a, i_0))$$

assuming $L_0 = R_0 = a$. In the recursive case, for any $p > 0$, $k = 2p$, the number of elements swapped is $N = k + 2$, and the equation is $L_{k+2} = R_{k+2}$, where

$$L_{k+2} = store(store(L_k, i_{k+1}, select(L_k, i_k)), i_k, select(L_k, i_{k+1}))$$
$$R_{k+2} = store(store(R_k, i_k, select(R_k, i_{k+1})), i_{k+1}, select(R_k, i_k)).$$

For example, for $N = 4$ ($k = 2$), we get $L_4 = R_4$ with

$$L_4 = store(store(L_2, i_3, select(L_2, i_2)), i_2, select(L_2, i_3))$$
$$R_4 = store(store(R_2, i_2, select(R_2, i_3)), i_3, select(R_2, i_2)).$$

For every N we get different instances by choosing different permutations of the operations, e.g., for $N = 4$, we can also generate $L'_4 = R'_4$, where $L'_4 = L_4$, and

$$R'_4 = store(store(R_2, i_3, select(R_2, i_2)), i_2, select(R_2, i_3)).$$

The above recursive definition only generates equations where all the pairs are exchanged. We also consider instances where only some of the pairs are exchanged. Thus, for N elements, there are $N!$ permutations, and $N!(2^{N/2} - 1)$ instances, where $2^{N/2} - 1$ is obtained from $\Sigma_{i=1}^{N/2} \binom{N/2}{i} = 2^{N/2} - 1$. Indeed, $\binom{N/2}{i}$ is the number of i-combinations over the set of $N/2$ pairs, or the number of ways of picking i pairs (for exchanging them) out of $N/2$. This expression counts each equation twice (e.g., $L = R$ and $R = L$), so that the number of distinct instances is $1/2(N!(2^{N/2} - 1))$. In our experiments, for each value of N, we sampled at most 16 permutations and 20 instances. We use $swap(N)$ to denote both the set of generated instances and the corresponding problem.

4 The experiments with E and CVC

We ran E (version 0.62 "Mullootar") and CVC on a dual AMD Athlon 1.2GHz machine, with 512MB of RAM, running Linux 2.4.7. We used both CVC/GRASP and CVC/Chaff, because they are two different versions of CVC, the latter released later. The SAT solver should not play a role with the theory of arrays. For all experiments, we gave precedence $select \succ store \succ sk$, completed by E by making all constant symbols smaller than the function symbols. As an exercise, we tried eight problems in the theory of arrays from the SVC distribution (SVC was CVC's predecessor: `http://sprout.Stanford.EDU/SVC/`). E in automatic mode solved each problem in 0.01 sec or less, with or without flattening, and CVC did each problem in approximately 0.04 sec.

We wrote a Prolog program that, given N, generates the instances of $storecomm(N)$ and $swap(N)$, in either E-LOP syntax, the Prolog-like syntax of E, or CVC syntax; it applies flattening for the experiments with E, and pre-processes the generated equations with respect to extensionality, for the experiments with E and T_1. Different instances of the same problem may have different numbers of distinct subterms: since the latter determines the number of new constants introduced by flattening, and each new constant is defined by an equation, different instances yield sets of equations of different size. The reported performance of a system on $storecomm(N)$, or $swap(N)$, is the *average* performance over all generated instances for size N.

We start with $storecomm(N)$, with presentation T_1 and strict flattening for the input to E. In Figure 1, E-Auto refers to E in automatic mode, while E-SOS refers to the plan `(SimulateSOS,Refinedweight(2,1,2,1,1))`, with selection function `SelectComplex` and ordering LPO, which was

among the best we observed. The curve for E-SOS is just a bit above those for CVC/GRASP and CVC/Chaff up to $N = 110$, then crosses them, and stays clearly below them for $N > 110$. The curve for E-Auto remains above the others, and is very smooth, which appears a welcome sign of regularity, considering how theorem provers have been often considered very sensitive to even minor input variations. Altogether, the theorem prover fared very well.

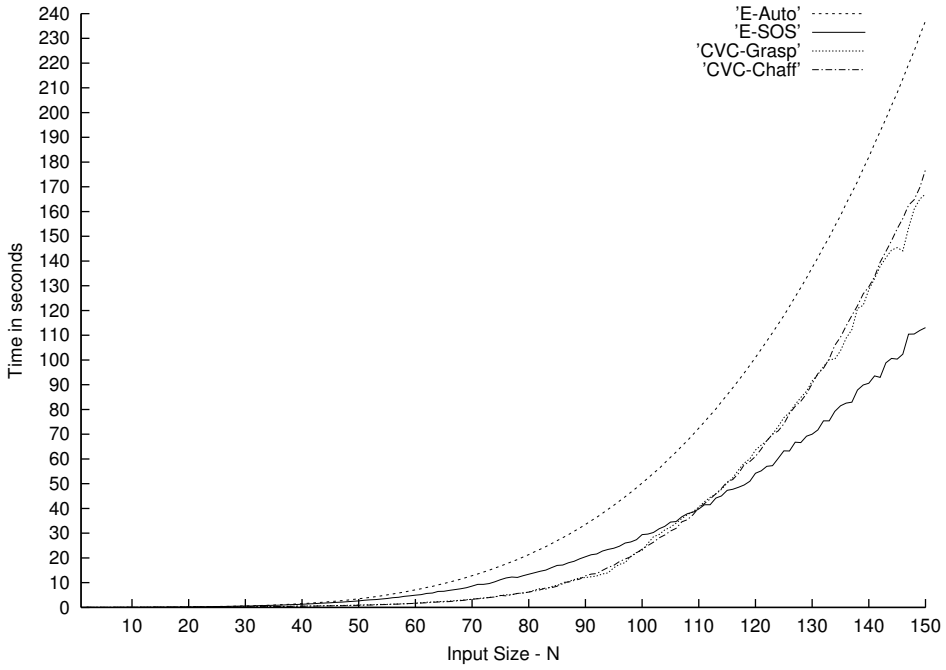


Figure 1: Behavior of E, with presentation T_1, and CVC on $storecomm(N)$, for N ranging from 2 to 150.

For $swap(N)$, we report the data in Table 1, because N ranges only from 2 to 10, since both E, with presentation T_1, and CVC, with either GRASP or Chaff, ran out of memory on any instance of $swap(12)$. We tried both strict and non-strict flattening, and E did best with the latter and a slight modification of the above search plan: `(SimulateSOS,Refinedweight(3,2,3,2,1))` with `SelectComplex` and KBO. With the exception of $swap(2)$, CVC performed better than E by one order of magnitude. The outcome is strikingly different, however, if we give T_2 in input to E, while using strict flattening. Figure 2 compares the performance of E on this input with that of CVC-Chaff from Table 1. Both E-Auto and E-SOS terminate successfully also for $N \geq 12$: the curve for E-SOS (with LPO and weights as for $storecomm(N)$) grows extremely slowly, while that for E-Auto is much higher but still smooth for the most part.

When we submitted in error redundant versions of $storecomm(N)$, E surpassed CVC sooner (in Figure 1, the E-SOS curve was below the CVC curves for $N \geq 60$ instead of $N \geq 110$). This suggests that E may be better than CVC in deleting redundant data. The algorithm of [19], in essence, pre-processes the input problem with respect to the axioms in T_1, eliminates the occurrences of *store* by recurring to *partial equations*, and computes a congruence closure. Thus, there might not be a provision to eliminate redundant equations, as theorem provers do by contraction. For E, the presentation T_2 may represent more information than T_1 with pre-processing with respect to extensionality, so that the prover behaves better on $swap(N)$, even if

N	E-Auto	E-SOS	CVC-GRASP	CVC-Chaff
2	0.010	0.010	0.043	0.057
4	0.144	0.136	0.059	0.060
6	2.205	2.110	0.189	0.187
8	63.630	62.230	4.091	2.400
10	2069.700	2039.700	1297.000	95.780

Table 1: Behavior of E with presentation T_1 and CVC on $swap(N)$.

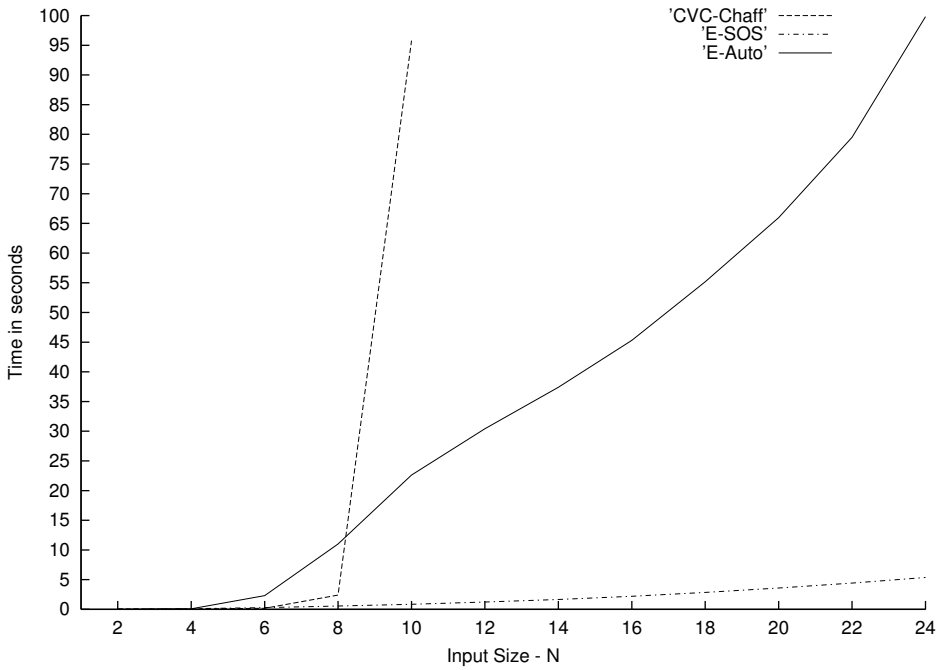


Figure 2: Behavior of E, with presentation T_2, and CVC on $swap(N)$, for N ranging from 2 to 24.

T_2 is not saturated and the prover is acting as a semi-decision procedure. We regard this behavior as evidence of the flexibility of an approach based on general-purpose deduction. Both E and CVC come with *proof checkers*. The E distribution features two tools, *e2pcl*, to extract a proof object from E's output, and *checkproof*, to check it by using another prover. While we did not use *flea*, the proof checker for CVC [20], we tried *e2pcl*, *checkproof* and Otter on sample outputs of E, and no error was detected.

5 Discussion

We tested the usage of theorem-proving strategies as decision procedures, on synthetic benchmarks in the theory of arrays with extensionality. The results indicate that this investigation should continue, and we envision several directions, in experimentation, implementation, and theory. For experimentation, we intend to work with more synthetic benchmarks (e.g., in the theory of

extensional finite sets, also covered in [1]), real-world problems (e.g., completing those in [4], already successfully started in [1]), problems involving other theories (e.g., for other data-types [13]) and *combinations of theories*. We also plan to conduct more experiments to understand the role of *flattening* better. Flattening aids by inducing a fully shared representation of terms as *dags*, which has been traditionally considered an advantage of congruence closure, and by making terms *shallow*, while increasing the number of equations. The first factor might not play such a key role for E, however, since E represents terms internally as *perfectly shared terms* regardless of the input [15]. The second factor might, since shallow terms simplify *matching* and all *term indexing* operations. The impact of the additional equations should be negligible, since provers are designed to handle millions, and a prover that inter-reduces its input uses only then an equation reducing a complex term to a constant. Surprisingly, and unlike Otter, E does not inter-reduce its input before starting the search.

It would be interesting to try other provers, especially those that implement the Otter version of the given-clause loop, to see whether a broader application of *eager contraction* helps in these problems. In our experiments, the difference between automatic mode and user-selected search plan had a visible impact on asymptotic behavior. This, together with the need of reducing the time spent testing search plans, invites more work on the *automatic mode* of provers, and more attention to *search plan design*. We felt at times that architecture and presentation of contemporary provers overemphasize blind saturation at the expense of search control.

Directions for theoretical research include termination and complexity results for more decidable theories. Satisfiability of a conjunction of literals in the theory of arrays with extensionality is NP-complete [19]. The algorithm of [19] has worst-case time complexity $O(2^{n \log n})$, where n is the size of the set of literals. For the deduction-based approach, the upper bound on the number of clauses that can be generated from $T \cup S$, where T contains the first two axioms of the theory and S is a set of flat equational literals, pre-processed with respect to extensionality, is $O(2^{n^2})$ [1]. However, this analysis refers to saturation, and does not take any search plan into account. The *complexity of theorem-proving strategies*, defined as the combination of inference system and search plan, is still largely unexplored, primarily because the underlying problem is only semi-decidable in the general first-order case. Results such as those of [1] exclude infinite derivations, and open the way to studying the complexity of concrete theorem-proving strategies for specific decidable theories. For theories where termination of saturation may not be proved, one may investigate obtaining a decision procedure by *integrating* theorem proving and model building. Indeed, the perspective of *system integration* encompasses all these directions: since one of the motivations for studying decision procedures is to integrate them into proof assistants, using theorem-proving strategies as decision procedures goes in the direction of fostering the integration of proof assistants and theorem provers.

Acknowledgements We would like to thank Stephan Schulz, for making E available and answering our questions on his prover, and the anonymous referees for their comments.

References

[1] Alessandro Armando, Silvio Ranise, and Michaël Rusinowitch. A rewriting approach to satisfiability procedures. *Information and Computation*, to appear, 2002.

[2] Leo Bachmair, Ashish Tiwari, and Laurent Vigneron. Abstract congruence closure. *Journal of Automated Reasoning*, to appear, 2002.

[3] Clark W. Barrett, David L. Dill, and Aaron Stump. A framework for cooperating decision procedures. In David McAllester, editor, *Proc. CADE-17*, volume 1831 of *LNAI*, pages 79–97. Springer, 2000.

[4] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In David L. Dill, editor, *Proc. CAV-6*, volume 818 of *LNCS*, pages 68–80. Springer, 1994.

[5] Graham Collins and Donald Syme. A theory of finite maps. In *Proc. TPHOLs*, LNCS. Springer, 1995.

[6] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–320. Elsevier, 1990.

[7] D. L. Detlefs, G. Nelson, and J. Saxe. Simplify: the ESC Theorem Prover. Technical report, DEC, 1996.

[8] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, Eds. *Computer Aided Reasoning : ACL2 Case Studies*. Kluwer, 2000.

[9] William W. McCune. Otter 3.0 reference manual and guide. Technical Report 94/6, MCS Division, Argonne National Laboratory, 1994. See also the web page `http://www-unix.mcs.anl.gov/AR/otter/`.

[10] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proc. 39th Design Automation Conf.*, 2001.

[11] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM TOPLAS*, 1(2):245–257, 1979.

[12] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: a prototype verification system. In Deepak Kapur, editor, *Proc. CADE-11*, volume 607 of *LNAI*, pages 748–752. Springer, 1992.

[13] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998. See also the web page `http://www-2.cs.cmu.edu/afs/cs.cmu.edu/user/jcr/www/`.

[14] Harald Rueß and Natarajan Shankar. Deconstructing Shostak. In *Proc. LICS-16*. IEEE, 2001.

[15] Stephan Schulz. E – a brainiac theorem prover. *AI Communications*, 2002. See also the web page `http://wwwjessen.informatik.tu-muenchen.de/~schulz/WORK/eprover.html`.

[16] Robert E. Shostak. Deciding combinations of theories. *J. ACM*, 31(1):1–12, 1984.

[17] João Marques Silva and Karem A. Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[18] Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: a Cooperating Validity Checker. In Kim G. Larsen and Ed Brinksma, editors, *Proc. CAV-14*, volume to appear of *LNCS*. Springer, 2002. See also the web page `http://verify.stanford.edu/CVC/`.

[19] Aaron Stump, Clark W. Barrett, David L. Dill, and Jeremy Levitt. A decision procedure for an extensional theory of arrays. In *Proc. LICS-16*. IEEE, 2001.

[20] Aaron Stump and David L. Dill. Faster proof checking in the Edinburgh Logical Framework. In Andrei Voronkov, editor, *Proc. CADE-18*, volume to appear of *LNAI*. Springer, 2002.

Translating the Object Constraint Language into First-order Predicate Logic

Bernhard Beckert, Uwe Keller, and Peter H. Schmitt

Universität Karlsruhe
Institut für Logik, Komplexität und Deduktionssysteme
Am Fasanengarten 5, D-76128 Karlsruhe
Fax: +49 721 608 4211, Email: {beckert,keller,pschmitt}@ira.uka.de

Abstract. In this paper, we define a translation of UML class diagrams with OCL constraints into first-order predicate logic. The goal is logical reasoning about UML models, realized by an interactive theorem prover. We put an emphasis on usability of the formulas resulting from the translation, and we have developed optimisations and heuristics to enhance the efficiency of the theorem proving process.
The translation has been implemented as part of the KeY system, but our implementation can also be used stand-alone.

1 Introduction

Overview. The Unified Modeling Language (UML) [15] has been widely accepted as the standard object-oriented modelling language and is supported by a great number of CASE tools. The Object Constraint Language (OCL) is an integral part of UML, and was introduced to express subtleties and nuances of meaning that diagrams cannot convey by themselves.

There is by now a great number of papers attributing a rigorous meaning to UML class diagrams (without OCL constraints) by translating them into a language with known semantics, for example: the CASL-LTL language (an extension of CASL) [16], the Z specification language [6] and its extension Object Z [13], the logical language of PVS [14], the Mathematical System Model (MSM) [5], EER diagrams [7], the Maude language [2].

Clarification of the semantics of UML class diagrams, as provided by these papers, is a necessary prerequisite for a rigorous semantics of OCL, as e.g. developed in [8, 9, 17] and in the draft [4]. We believe that the semantics of UML class diagrams with OCL, both the issues of common consent and controversial open issues, are by now understood well enough to serve as a basis for further developments. The translation developed in this paper can be applied to OCL constraints in any UML diagram type. But since the semantical status of OCL constraints in other diagram types, such as state or sequence diagrams, is less clear, we restrict attention for the moment to OCL constraints in class diagrams.

We present in this paper a translation of UML/OCL into first-order predicate logic. The translation covers the complete OCL language definition, except that we do not support the three valued logic of OCL. Further, minor restrictions, will be mentioned in the body of the paper.

Our goal is logical reasoning about UML models. The novel features of our work are that we put an emphasis on usability of the formulas resulting from the translation, and we offer alternatives for the translation of model elements. Where possible, we develop optimisations and heuristics to enhance the efficiency of the theorem proving process. For interactive theorem proving ease of use for the interacting human prover is a central factor for efficiency. Therefore readability of the translated formulas becomes a crucial issue.

The KeY Project. The work reported here is part of the KeY project (see the overview paper [1] or the web page `i12www.ira.uka.de/~key` for more information). The logical language used in this project is Dynamic Logic, a multi-modal extension of first-order predicate logic specially suited to reason about properties of programs. In this present account we restrict attention to translation into first-order logic, which is the crucial part anyhow. The extension of the

translation to the OCL constructs that require Dynamic Logic as the target language, e.g. @*pre* and `result` in post-conditions and the `iterate` operation, is rather straightforward and can be found in [12]. An extensive account of how to treat the @*pre* operator in Dynamic Logic is given in [3].

Implementation. We have implemented our translation, including some optimisations and heuristics. The implementation, which is written in JAVA, is part of the KeY system and works automatically without user interaction. For those who wish to use the translation in a different context, we have provided a stand-alone version that reads the UML class diagram and the OCL constraints to be translated from an XML file and generates a text file containing the resulting formulas. It uses the XML dialect XMI, which is a standard for the textual representation of UML diagrams. For parsing OCL constraints, we have integrated the parser component of the OCL compiler developed by Hußmann et al. [11].

We tried to keep the implementation flexible and it should be easy to adapt to different needs arising from other application areas, such as a different syntax for the output formulas, new optimisations, and new heuristics for choosing between several possible translations. Also, adaptations to future changes in the UML/OCL standard will not require much effort.

Both the KeY system and the stand-alone version, as well as additional documentation and examples, can be downloaded from `i12www.ira.uka.de/~key`.

To the best of our knowledge, this is the first implementation of such a translation. We hope that it can serve as a means helping to promote the use and application of OCL.

Structure of this Paper. In Section 2, we briefly review the semantical pre-requisites and describe the semantical properties of our translation. The basic translation is presented in Sections 3 and 4, while Section 5 is devoted to possible optimisations that improve the readability and usability of the resulting first-order formulas. Section 6 concludes with an outlook and future extensions.

All examples presented in the following refer to the class diagram shown in Figure 1.

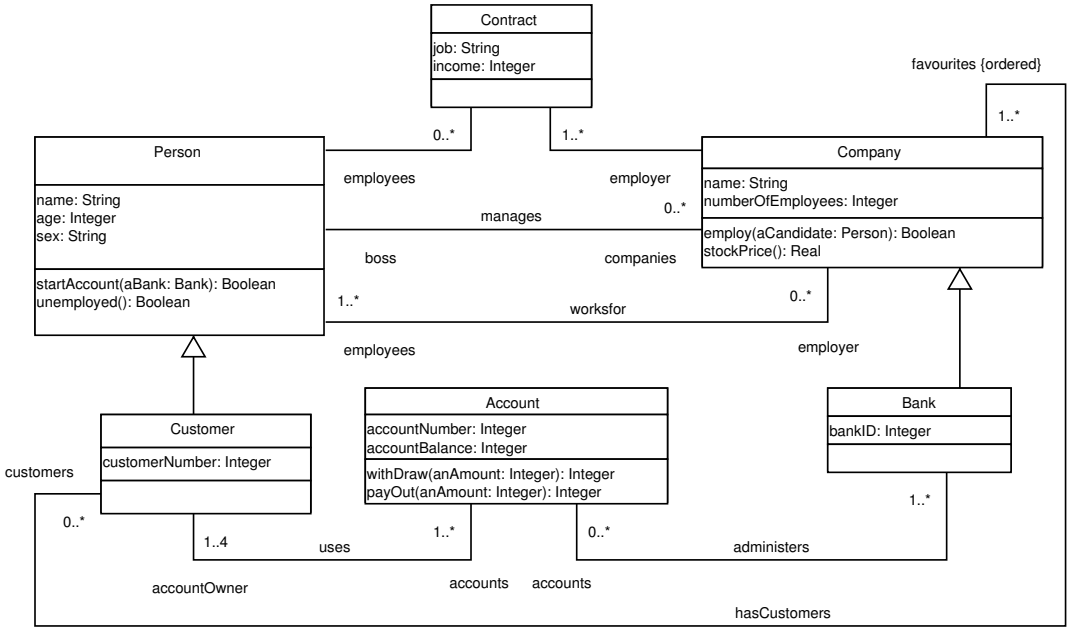


Fig. 1. Example for a UML class diagram.

2 Properties of the Translation

We start with a given UML class diagram $\mathcal{D}$ that is enriched by OCL constraints $C_1, \ldots, C_n$. Together, $\mathcal{D}$ and $C_1, \ldots, C_n$ describe the possible *states* of the system to be modelled. A system state, sometimes also called a *snapshot* in the UML framework, is a complete description of an instance of the modelled system. It details what objects exist (they are instances of the classes in $\mathcal{D}$), gives the values of attributes for the existing objects, and defines which pairs of objects (or more general, n-tuples of objects) are instances of the associations between classes in $\mathcal{D}$. We use first-order structures $\mathcal{S}$ to represent system states.

The vocabulary $\Sigma = \Sigma_{\mathcal{D}}$ of $\mathcal{S}$, i.e., the set of types, function, and relation symbols, is read off from the diagram $\mathcal{D}$. Sometimes there are choices in which symbols to include in $\Sigma_{\mathcal{D}}$: A binary association between classes A and B with multiplicity 1 at the B-end may give rise to the inclusion of a binary relation symbol in $\Sigma_{\mathcal{D}}$ or of a unary function symbol. To have a common platform for comparing these alternatives, we include (in this and similar cases) both symbols in $\Sigma_{\mathcal{D}}$. The definition of $\Sigma_{\mathcal{D}}$ follows shortly.

Of course, not all $\Sigma_{\mathcal{D}}$-structures are valid system states of $\mathcal{D}$. We will say that a structure $\mathcal{S}$ *conforms* to $\mathcal{D}$ in case that $\mathcal{S}$ satisfies the diagram $\mathcal{D}$ and its OCL constraints $C_1, \ldots, C_n$, i.e., it is a possible system state according to the UML/OCL semantics [15].

In the next two sections, we describe how to associate with a UML class diagram $\mathcal{D}$ and OCL constraints $C_1, \ldots, C_n$ formulas $Th_{\mathcal{D}}, Th_{C_1}, \ldots, Th_{C_n}$. Since new symbols are added by the translation, they are formulas over an extended signature $\Sigma^* = \Sigma_{\mathcal{D}} \cup \Sigma_{tr}$. Therefore, the correctness property of our translation reads: For every $\Sigma_{\mathcal{D}}$-structure $\mathcal{S}$,

$$\begin{array}{l} \mathcal{S} \text{ conforms to } \mathcal{D} \text{ with } C_1, \ldots, C_n \quad \text{if and only if} \\ \mathcal{S}^* \models Th_{\mathcal{D}} \wedge Th_{C_1} \wedge \ldots \wedge Th_{C_n} \text{ for every } \Sigma^*\text{-extension } \mathcal{S}^* \text{ of } \mathcal{S}. \end{array}$$

A detailed analysis of the correctness property of such a translation can be found in [12].

Our translation does not handle meta level features—with the exception of `OCLAny` and `allInstances`. This is due to that fact, that the future role of the meta level is unclear. In version 2.0 of the UML standard, now under discussion, it may undergo substantial changes or be eliminated alltogether.

3 Translating the Class Diagram

3.1 Extracting the Signature from the Class Diagram

In the following, we summarise how the first-order signature $\Sigma_{\mathcal{D}}$ is extracted from a class diagram $\mathcal{D}$. A more extensive account may be found in [18]. The set of *types* of $\Sigma_{\mathcal{D}}$ contains:

1. A type for every class in $\mathcal{D}$. Types will be denoted by the same names as the corresponding class, starting with an upper-case letter.
2. The types *Integer*, *Real*, *Boolean*, *String*.
3. If T is a type, then $Collection_T$, Set_T, Bag_T, $Sequence_T$ are types. Types of this form are called *collection types*. These collection types are only generated when T is not itself a collection type, i.e., no nesting of the collection type operators is allowed.
4. $\Sigma_{\mathcal{D}}$ will furthermore contain the type *Any*, which serves as the translation of the OCL type *OCLAny*.

The subtype relation $S_1 <_{\mathcal{D}} S_2$ is defined as in [19]. For each type T there will be an infinite supply of variables $x{:}T, y{:}T, x_i{:}T$ of type T. The set of *functions* and *relations* in $\Sigma_{\mathcal{D}}$ contains:

1. For every binary association r in $\mathcal{D}$ with association ends e_0, e_1 there are two functions in $\Sigma_{\mathcal{D}}$, which will be referred to by the role name r_i at the association end e_i $(i = 0, 1)$. If no role name is given, the name of the class attached to e_i will be used. Function names start with a lower-case letter. If e_i is attached to class S_i, then the function is of signature $r_i \colon S_{1-i} \to Set_{S_i}$. In case the multiplicity at the end e_i is 1, the signature is $r_i \colon S_{1-i} \to S_i$. If the e_i-end has the stereotype $\ll ordered \gg$, then it is $r_i \colon S_{1-i} \to Sequence_{S_i}$. For n-ary relations we proceed correspondingly.

2. For every n-ary association r in $\mathcal{D}$ there is, in addition, an n-ary *predicate* in $\Sigma_\mathcal{D}$.
3. For every *attribute* a of a class S in $\mathcal{D}$ there is a function in $\Sigma_\mathcal{D}$ that is referred to by the name of the attribute and has signature $a\colon S \to S_r$, where S_r is the value type of attribute a as specified in $\mathcal{D}$. If a is a *class* attribute (sometimes this is also called a static attribute), then a constant of type S_r is added to $\Sigma_\mathcal{D}$. The concrete syntax of this constant is $S.a$.
4. For every *operation* c of a class S with parameters of type $S_1, \ldots, S_k$ and result type S' there is a function $f_c : S \times S_1 \times \ldots \times S_k \to S'$ in $\Sigma_\mathcal{D}$. In accordance with the OCL specification [15] we require that c has no side effects, i.e., it satisfies the property `isQuery()`.
5. For every *association class* C attached to an association r, where r associates the classes S_1 and S_2, there are unary projection functions $s_1\colon C \to S_1$ and $s_2 : C \to S_2$ in $\Sigma_\mathcal{D}$.
6. All properties of the pre-defined OCL types, as detailed in the standard [15], are functions or relations in $\Sigma_\mathcal{D}$.
7. The symbol $\doteq$ will be used to denote equality. By overloading we use the same symbol for all types.

3.2 Extracting Formulas from the Class Diagram

The translation $Th_\mathcal{D} = (\bigwedge Ax_{ADT} \wedge \bigwedge Ax_\mathcal{D} \wedge \bigwedge Constr_\mathcal{D})$ of a class diagram $\mathcal{D}$ consists of three parts: Ax_{ADT} is actually independent of $\mathcal{D}$. It contains the axioms of the Abstract Data Types (ADTs) that are used to represent the built-in data types of OCL (*Integer*, *Boolean*, etc.), and the axioms for the ADTs Set_T, Bag_T, etc. that are used to represent the corresponding collection types of OCL.

The second part $Ax_\mathcal{D}$ is a set of axioms that depend on $\mathcal{D}$ but that do not express intrinsic information of $\mathcal{D}$. They deal with inter-dependencies among the function and relation symbols extracted from $\mathcal{D}$ that reflect, for example, the symmetry of associations in UML.

Example 1. Consider the association *worksfor* between the classes *Person* and *Company* (Figure 1). The signature $\Sigma_\mathcal{D}$ contains two function symbols and a relation symbol representing this association: *employer* with argument type *Person* and value type $Set_{Company}$, *employees* with argument type *Company* and value type Set_{Person}, and the binary relation symbol *worksfor* with first argument of type *Person* and second argument of type *Company*.

To restrict the interpretation of these symbols appropriately, the set $Ax_\mathcal{D}$ contains the following axioms:

$$\forall p{:}Person\, \forall c{:}Company\, (c \in employer(p) \leftrightarrow p \in employees(c))$$
$$\forall p{:}Person\, \forall c{:}Company\, (c \in employer(p) \leftrightarrow worksfor(p, c))$$
$$\forall p{:}Person\, \forall c{:}Company\, (p \in employees(c) \leftrightarrow worksfor(p, c))$$

The third part $Constr_\mathcal{D}$ of $Th_\mathcal{D}$ contains formulas representing the restrictions on system states expressed graphically in $\mathcal{D}$, e.g. multiplicity constraints, subtyping restrictions, and others. We require in $Constr_\mathcal{D}$ also that an abstract class is the union of its concrete subclasses and that enumerations are sets containing exactly their enumeration literals as elements. A detailed account of this topic is given in [12]. Instead of giving a formal definition of $Constr_\mathcal{D}$, we present a typical example:

Example 2. Consider the association *uses* between the classes *Customer* and *Account*. The set $Constr_\mathcal{D}$ contains the following formulas expressing the multiplicity constraints attached to *uses*:

$$\forall c{:}Customer\, (size(accounts(c)) \geq 1)$$
$$\forall a{:}Account\, (1 \leq size(accountOwner(a)) \wedge size(accountOwner(a)) \leq 4)$$

4 Translating the OCL Constraints

4.1 Overview

OCL constraints consist of an OCL expression of type *Boolean* and some declaration connecting the OCL expression to an item in the class diagram. In the case of pre- and post-conditions, the constraint is bound to an operation; invariants are bound to a class. The translation procedure for OCL constraints, therefore, cannot process OCL expressions as *isolated* entities but also has to take into account the diagram and the information it contains.

In our basic translation described below, OCL expressions in most cases are translated into a first-order *term* of the appropriate Abstract Data Type (ADT). The only exceptions are expressions of OCL type *Boolean*, which are usually transformed into first-order *formulas*. The first-order term resp. formula that is the result of translating an OCL expression exp is denoted by $\lceil exp \rceil$.

The translation procedure works by structural recursion on the expressions. When certain OCL features are translated (as described in the following subsections), new function or predicate symbols are introduced (they are elements of the extended signature Σ^*) as well as axioms that constrain the interpretation of the introduced symbols according to the semantics of UML/OCL.

Note, that OCL allows a modeller to use some shorthand notations. We assume that constraints have been normalised to their (longer) standard form before they are translated (in our implementation we use a normalisation provided by Hußmann et al.'s OCL compiler [11]).

The set Ax_{exp} generated during the translation of an expression exp includes all those axioms that are generated by the recursive translation of subexpressions of exp—besides the axioms that stem from the translation of the "top-level" OCL feature of exp.

The translation of OCL *expressions* is extended to OCL *constraints* as follows. Let I be an OCL invariant of form "`context` C `inv:` b", where C is a class in the diagram $\mathcal{D}$ and b is an OCL expression of type *Boolean*. The invariant states that, for *every* instance *self* of C existing in a system state, the property described by b holds. Accordingly, the translation Th_I of the invariant I is:

$$\bigwedge Ax_b \rightarrow \forall self{:}C\ \lceil b \rceil \ .$$

Pre- and post-conditions can be translated in a similar way; only the @*pre* operator, which may occur in post-conditions, requires a special treatment (see [3] and [12] for a detailed account).

4.2 Translating Built-in OCL Types

Translating Boolean Expressions. As said above, we usually translate OCL expressions of type *Boolean* into first-order formulas. The boolean operators `and`, `or`, `implies`, `not` are translated into the corresponding first-order operators. Equality of *Boolean* expressions is represented by the operator $\leftrightarrow$.

Translating Integer, Real, String Expressions. The OCL type *Integer* corresponds to an ADT *Integer*. As said above, the signature $\Sigma_{\mathcal{D}}$ contains a symbol for every feature[1] of *Integer*. Every feature of *Integer* is translated into the corresponding function or predicate symbol of the ADT. In the same way, the OCL types *Real* and *String* are handled with the help of ADTs *Real* and *String*.

Hußmann et al. [10] have argued convincingly that the encapsulation concepts of ADTs and UML classes are very different and that UML classes can, as a consequence, not smoothly be translated into ADTs. However, their analysis applies primarily to user defined classes and does not affect the translation of the basic OCL types just mentioned.

Example 3. Given the following OCL expression (with respect to class *Person*)

[1] According to OCL terminology a *feature* of some OCL type T is any operation that can be applied to instances of T.

```
self.age >= 0 and self.employer->size >= 1
```

that states that person *self* is employed and has a non-negative age the translation results in the formula $age(self) \geq 0 \wedge size(employer(self)) \geq 1$.

4.3 Translating the allInstances Operator

The OCL operator `allInstances` can be applied to a class (to be more precise it is applied to the object of type *OclType* that corresponds to the class in the diagram $\mathcal{D}$). It returns the set of all instances of that class in the current state. To translate this operator, we introduce a new symbol $allInstances_C$ for each class C and define $\lceil C\texttt{.allInstances}\rceil = allInstances_C$. The additional axiom $\forall o{:}C\,(o \in allInstances_C)$ is introduced to specify the meaning of the new constant.

4.4 Translating Collection Operators

Overview. OCL offers a common super-type $Collection(T)$ for the collection types $Set(T)$, $Bag(T)$, and $Sequence(T)$. Since OCL defines this super-type to be abstract, it does not occur in actual OCL constraints, but is used to define features that all collection types have in common (e.g., the `size` operator). Consequently, we provide ADTs to represent sets, bags, sequences and collections of all occurring types (e.g., Set_{Bank}, Bag_{Bank}, $Sequence_{Bank}$, $Collection_{Bank}$), where the ADTs $Collection_T$ are only relevant for the purpose of typing in pathological borderline situations and play no role for modelling with OCL in practice.

Below, we describe the translation of the features that the collection types have in common.

Translating `size`, `count`, `sum`, `includes`, `append`, *etc.* These features are translated into the functions that are their direct counterparts in the ADTs Set_T, Bag_T, and $Sequence_T$. For example, $\lceil c\texttt{->size}\rceil = size(\lceil c\rceil)$ and $\lceil s\texttt{->union(}c\texttt{)}\rceil = union(\lceil s\rceil, \lceil c\rceil)$.

Translating Equality. We translate the equality $s_1\texttt{=}s_2$ of sets s_1, s_2 of OCL type $Set(T)$ by expressing that they have the same elements: $\lceil s_1\texttt{=}s_2\rceil = \forall e{:}T\,(e \in \lceil s_1\rceil \leftrightarrow e \in \lceil s_2\rceil)$. Here and in all similar situations below, $e{:}T$ is a new variable that has not been used before.

For bags we get the formula $\lceil b_1\texttt{=}b_2\rceil = \forall e{:}T\,(count(\lceil b_1\rceil, e) \doteq count(\lceil b_2\rceil, e))$, and for sequences a similar translation is generated.

Translating `includesAll`, `excludesAll`. $E = c_1\texttt{->includesAll(}c_2\texttt{)}$ expresses that the collection c_2 is a subset of c_1. Thus, $\lceil E\rceil = \forall e{:}T\,(e \in \lceil c_2\rceil \rightarrow e \in \lceil c_1\rceil)$. `excludesAll` expresses that no element of c_2 is an element of c_1 and is treated similarly.

Translating `notEmpty`, `isEmpty`. The translation of the expression $E = c\texttt{->notEmpty}$ is the formula $\lceil E\rceil = \exists e{:}T\,(e \in \lceil c\rceil)$. `isEmpty` is treated as the negation of `notEmpty`.

Translating `forAll`, `exists`. The meaning of $E_1 = c\texttt{->forAll(e| }b\texttt{)}$ is that b evaluates to `true` for all possible instantiations of `e` with elements of the collection c. Thus, the translation of E_1 is $\lceil E_1\rceil = \forall e{:}T\,((e \in \lceil c\rceil) \rightarrow \lceil b\rceil)$. To translate $E_2 = c\texttt{->exists(e| }b\texttt{)}$ we use $\lceil E_2\rceil = \exists e{:}T\,((e \in \lceil c\rceil) \wedge \lceil b\rceil)$.

Example 4. Consider the following OCL expression, which formalises "For different objects of class *Bank*, the attribute *bankID* has different values."

```
Bank.allInstances->forAll(b1,b2 |
       not (b1 = b2)  implies  not (b1.bankID = b2.bankID))
```

Its translation is the following formula (a much shorter and optimised translation is given in Section 5.3):

Translation:

$$\forall b_1{:}Bank\,(b_1 \in allInstances_{Bank} \rightarrow \forall b_2{:}Bank\,(b_2 \in allInstances_{Bank} \rightarrow (\neg(b_1 \doteq b_2) \rightarrow \neg(bankID(b_1) \doteq bankID(b_2)))))$$

Additional axiom:

$$\forall b{:}Bank\,(b \in allInstances_{Bank})$$

Translating `isUnique`. The meaning of $E = c\texttt{->isUnique(e|}exp\texttt{)}$ is that the evaluation of exp results in a different value for each instantiation of `e` with elements of c. So,

$$\lceil E \rceil \;=\; \forall e_1{:}T\, \forall e_2{:}T\, ((e_1 \in \lceil c \rceil \wedge e_2 \in \lceil c \rceil \wedge \lceil exp \rceil\{e/e_1\} \doteq \lceil exp \rceil\{e/e_2\}) \;\rightarrow\; e_1 \doteq e_2)\;.^{2}$$

where $e_1{:}T, e_2{:}T$ are two distinct new variables.

Translating `sortedBy`. The value of $E = c\texttt{->sortedBy(e|}exp\texttt{)}$ is a sequence with (a) the same elements as collection c, which are (b) ordered according to the values of the expression exp (this only makes sense if there is some order $\leq$ defined on the OCL type of exp. To translate E, we introduce a new function symbol $sortedBy_E$. Let $p_1, \ldots, p_n$ be the free variables occurring in the translations $\lceil c \rceil$ and $\lceil exp \rceil$ of the subexpressions—excluding the variable `e`. Then, the translation of E is $\lceil E \rceil = sortedBy_E(p_1, \ldots, p_n)$. To ensure that $sortedBy_E$ has the desired interpretation with properties (a) and (b), the following two axioms are added to Ax_E:

$$\forall p_1{:}T_1 \ldots \forall p_n{:}T_n \forall e'{:}T\, (count(\lceil \texttt{c} \rceil, e') \doteq count(sortedBy_E(p_1, \ldots, p_n), e'))$$

$$\begin{aligned}&\forall p_1{:}T_1 \ldots \forall p_n{:}T_n \forall i{:}Integer, j{:}Integer\, (\\ &\quad (1 \leq i \wedge i \leq j \wedge j \leq size(sortedBy_E(p_1, \ldots, p_n))) \rightarrow \\ &\qquad \lceil \texttt{exp} \rceil\{e/at(sortedBy_E(p_1, \ldots, p_n), i)\} \\ &\qquad \leq \lceil \texttt{exp} \rceil\{e/at(sortedBy_E(p_1, \ldots, p_n), j)\})\end{aligned}$$

where $e'{:}T$ and $i{:}Integer, j{:}Integer$ are distinct new variables.

Translating `select`, `reject`. The expression $E = c\texttt{->select(e|}b\texttt{)}$ denotes the collection consisting of those elements of c for which b evaluates to `true` when `e` is instantiated with the element. The translation is based on introducing a new function symbol $select_E$. Let $p_1, \ldots, p_n$ be the free variables occurring in the translation $\lceil b \rceil$ of the condition b excluding `e`. Then, the translation of E is $\lceil E \rceil = select_E(\lceil \texttt{c} \rceil, p_1, \ldots, p_n)$.[3] Three axioms are added to specify the meaning of $select_E$. Their form depends on whether c is a set, a bag, or a sequence. Here, we present the axioms for sets (the axioms for the other types are similar):

$$\forall p_1{:}T_1 \ldots \forall p_n{:}T_n\, select_E(emptySet_T, p_1, \ldots, p_n) \doteq emptySet_T$$

$$\begin{aligned}&\forall p_1{:}T_1 \ldots \forall p_n{:}T_n \forall s{:}Set_T \forall e{:}T\, (\\ &\quad \lceil b \rceil \rightarrow select_E(insert(s, e), p_1, \ldots, p_n) \doteq insert(select_E(s, p_1, \ldots, p_n), e))\end{aligned}$$

$$\begin{aligned}&\forall p_1{:}T_1 \ldots \forall p_n{:}T_n \forall s{:}Set_T \forall e{:}T\, (\\ &\quad \neg\lceil b \rceil \rightarrow select_E(insert(s, e), p_1, \ldots, p_n) \doteq select_E(s, p_1, \ldots, p_n))\end{aligned}$$

Since the `reject` operator is just the opposite of `select`, we treat it by negating the filter condition b and then applying the above translation.

Example 5. The following OCL expression E formalises "There is no person who works for both company 'BankA' and company 'BankB'."

```
Person.allInstances->select(p| p.employer->exists(c1,c2 |
        c1.name = 'BankA' and c2.name = 'BankB'))->isEmpty
```

[2] The notation $t\{e/s\}$ denotes the result of syntactically replacing all occurrences of the variable e in the term t by the term s.

[3] In [9], a similar abbreviation technique is used for the translation of the `select` operator. There, however, the free variables $p_1, \ldots, p_n$ are not made arguments of the new function, which leads to incorrect results.

Its translation is the following formula (a much shorter and optimised translation is given in Section 5.3):

$$
\begin{array}{l}
\underline{\text{Translation:}} \\
\forall p{:}Person\,(\neg(p \in select_E(allInstances_{Person}))) \\
\\
\underline{\text{Additional axioms:}} \\
\forall p{:}Person\,(p \in allInstances_{Person}) \\
select_E(emptySet_{Person}) \doteq emptySet_{Person} \\
\forall s{:}Set_{Person}\,\forall p{:}Person\,(\exists c_1{:}Company\,(c_1 \in employer(p) \wedge \\
\qquad\qquad \exists c_2{:}Company\,(c_2 \in employer(p) \wedge \\
\qquad\qquad name(c_1) \doteq \text{`}BankA\text{`} \wedge name(c_2) \doteq \text{`}BankB\text{`})) \rightarrow \\
\qquad\qquad select_E(insert(s,p)) \doteq insert(select_E(s),p)) \\
\forall s{:}Set_{Person}\,\forall p{:}Person\,(\neg(\exists c_1{:}Company\,(c_1 \in employer(p) \wedge \\
\qquad\qquad \exists c_2{:}Company\,(c_2 \in employer(p) \wedge \\
\qquad\qquad name(c_1) \doteq \text{`}BankA\text{`} \wedge name(c_2) \doteq \text{`}BankB\text{`}))) \rightarrow \\
\qquad\qquad select_E(insert(s,p)) \doteq select_E(s))
\end{array}
$$

4.5 Translating Other Constructs of OCL

Translating `oclIsKindOf`, `oclIsTypeOf`. These operators allow to check which type the value of an expression exp has. The translation of exp`.oclIsKindOf(`T`)` is $\exists e{:}T\, e \doteq \lceil exp \rceil$. The operator `oclIsTypeOf` can be expressed by `oclIsKindOf` using the subtype relation extracted from the diagram $\mathcal{D}$.

Translating `oclAsType`. To translate the cast operator `oclAsType`, we introduce a new function symbol $oclAsType_{T_1,T_1} : T_1 \rightarrow T_2$ for every pair T_1, T_2 where T_2 is a subtype of T_1, and we define $\lceil o$`.oclAsType(`T_2`)`$\rceil = oclAsType_{T_1,T_2}(\lceil o \rceil)$ (where o is of type T_1). The additional axioms specifying these symbols are of the form $\forall x{:}T_2\,(oclAsType_{T_1,T_2}(x) \doteq x)$.

Translating Variables and Literals. The translation of an OCL variable `v`, including `self`, is a first-order variable with the same name, i.e., $\lceil$`v`$\rceil = v$.

OCL literals of type *Boolean*, *Integer*, *Real*, or *String* are translated into a term over the corresponding ADT. To translate literals of collection types, in case they enumerate the elements of a collection, we construct a term over the ADT Set_T (resp. Bag_T or $Sequence_T$). For example,

$$\lceil \texttt{Set\{1,2,3\}} \rceil = insert(insert(insert(emptySet_{Integer}, 1), 2), 3)\ .$$

To translate collection literals that specify a range of elements, such as $E =$ `Set{`e_1`..`e_2`}`, we introduce a new function symbol set_E and define $\lceil E \rceil = set_E(p_1, \ldots, p_n)$ (where $p_1, \ldots, p_n$ are the free variables occurring in the translations of the bounds e_1 and e_2). The additional axiom specifying set_E is

$$\forall p_1{:}T_1 \ldots \forall p_n{:}T_n\,\forall i{:}T\,(i \in set_E(p_1, \ldots, p_n) \leftrightarrow (\lceil e_1 \rceil \leq i \wedge i \leq \lceil e_2 \rceil))\ .$$

For bags and sequences, the translation is similar. However, additional axioms are needed to express that (a) every element in the range occurs exactly once in the result and (b) for sequences, that the elements are ordered.

Example 6. The following OCL expression (used as an invariant for class *Customer*) formalises "A customer's favourite companies are ordered according to their stock price."

```
Sequence {1 .. self.favourites->size}->forAll(i,j| j >= i   implies
   self.favourites->at(i).stockPrice()  >=
   self.favourites->at(j).stockPrice())
```

Its translation is the following formula (a much shorter and optimised translation is given in Section 5.3):

Translation:

$$\forall i{:}Integer\,(i \in seq_0(self) \rightarrow \forall j{:}Integer\,(j \in seq_0(self) \rightarrow (j \geq i \rightarrow stockPrice(at(favourites(self), i)) \geq stockPrice(at(favourites(self), j)))))$$

Additional axioms:

$$\forall c{:}Customer\,\forall i{:}Integer\,(i \in seq_0(c) \leftrightarrow 1 \leq i \wedge i \leq size(favourites(c)))$$

$$\forall c{:}Customer\,\forall i{:}Integer, j{:}Integer\,(1 \leq i \wedge i \leq j \wedge j \leq size(seq_0(c)) \rightarrow at(seq_0(c), i) \leq at(seq_0(c), j))$$

$$\forall c{:}Customer\,\forall i{:}Integer\,(count(seq_0(c), i) \leq 1)$$

5 Optimisations and Simplifications

5.1 Motivation

It is crucial for the usability of the formulas generated by the translation (in particular in *interactive* theorem proving)—and for the usefulness of such a translation itself—that the formulas are as easy to understand as the original OCL expressions. One way for achieving this goal is to generate a formula that is syntactically as close as possible to the translated OCL expression. For example, the names of the function symbols used in a formula should as far as possible coincide with the names of the corresponding features in OCL. We tried to satisfy this demand with our basic translation described in Section 4.

But, although the generated formulas are very similar to the original expression in their syntactic structure, they are sometimes unnecessarily complicated and hard to read. This is due to the additional axioms introduced in order to constrain the interpretation of new function symbols, which mainly represent OCL collections. Even for small OCL expression there can be a large number of constraining axioms.

A technique that aims to overcome this problem is presented in this section. The idea is to use a different representation for OCL collections. That can help to reduce the number of additional function symbols and axioms, because most of them are introduced when collection operators are translated (such as `select` and `asSequence`).

Note, that often the readability of formulas can be improved by applying rewriting rules. But there are also many cases where the effects of an unsuitable translation cannot be undone by mere simplification but where the form of the original constraint has to be known to choose a good first-order representation; such choices, consequently, have to be made at the time and as part of translation.

5.2 Representing Collections with Predicates

The translation described in Section 4 uses a functional representation of OCL collections, i.e., expressions of type *Set*, *Sequence*, or *Bag* are translated into first-order terms. An alternative is to translate such expressions predicatively, i.e., to represent them by a formula.

For *sets* a predicative translation is easy to define: A set expression s is translated into a "characteristic" formula $\phi_s(e)$ that is equivalent to $e \in \lceil s \rceil$. Using such a presentation allows us to translate most OCL set operators without the need to introduce new function symbols. For example, the expression $E = s$`->select(e|`b`)` can then be represented by $\phi_E = \phi_s(e) \wedge \lceil b \rceil$.

Unfortunately, there are also cases where a predicative representation is not useful. For example, when an expression of the form s`->size` is translated, it is better to apply a functional translation to the subexpression s. Also, for bags and sequences, a useful predicative representation is more difficult to define than for sets, and the resulting formulas are often hard to understand.

Our investigation of examples showed however, that a predicative translation of *sets* is preferable in most cases. Moreover, in many OCL constraints, expressions of type *bag* or *sequence* are actually used as sets, i.e., the additional information they contain is irrelevant. For example, when an expression $E = s$`->forAll(e|`b`)` is translated, the order of elements in s and the number of their occurrences is of no importance, and E can be translated predicatively. This basic idea gives rise to a simple but powerful heuristics to decide whether a predicative translation is preferable to a functional form. Usually such a decision has to be made globally for a whole expression, because combining the translations of subexpressions that use different representations (functional resp. predicative) is awkward and leads to formulas that are hard to read. For a detailed account of this topic please refer to [12].

5.3 Examples for Predicative Translations

In this section, we present the predicative translations of the OCL expression from the examples in Section 4. They are shorter and easier to read than the functional translations. Moreover, it is not necessary anymore to generate additional axioms since no new symbols are introduced.

Example 7. The predicative translation of the expression from Example 4 is:

$$\forall b_1{:}Bank\ \forall b_2{:}Bank\ (\neg(b_1 \doteq b_2) \rightarrow \neg(bankID(b_1) \doteq bankID(b_2)))$$

Example 8. The OCL expression from Example 5 translates to:

$$\begin{aligned}\forall p{:}Person\,(\neg\,(&\exists c_1{:}Company\,(c_1 \in employer(p) \wedge\\ &\exists c_2{:}Company\,(c_2 \in employer(p) \wedge\\ &name(c_1) \doteq \text{`}BankA\text{`} \wedge name(c_2) \doteq \text{`}BankB\text{`}))))\end{aligned}$$

Example 9. The predicative translation of the expression from Example 6 is:

$$\begin{aligned}\forall j{:}Integer\,(&1 \leq j \wedge j \leq size(favourites(self)) \rightarrow\\ &\forall i{:}Integer\,(1 \leq i \wedge i \leq size(favourites(self)) \rightarrow\\ &\quad (j \geq i \rightarrow\\ &\quad stockPrice(at(favourites(self), i)) \geq\\ &\quad stockPrice(at(favourites(self), j)))))\end{aligned}$$

6 Conclusions and Future Work

We have presented in this paper a translation of the logical information contained in UML class diagrams and OCL constraints into first-order predicate logic. It has been implemented as part of the KeY system. It should be easy to use it with other systems, since first-order logic by its universal nature can be readily mapped into almost all logical languages used in formal methods.

We have provided a first set of optimisations. Experimenting with case studies will give insight if and which further optimisations are necessary or desirable. In the present implementation the optimisations are chosen by a fixed built-in heuristic. It would be easy to extend the implementation to allow custom-made heuristics.

In the present account, we have deliberately excluded some items, e.g. the `iterate` operator, that are better expressed in a higher-order logic. These issues are treated in [12].

It also remains future research to compare and possibly adapt our translation to version 2.0 of UML/OCL standard once it is agreed upon.

References

1. W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, and P. H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In M. Ojeda-Aciego, I. P. de Guzman, G. Brewka, and L. M. Pereira, editors, *Proceedings, Logics in Artificial Intelligence (JELIA), Malaga, Spain*, LNCS 1919. Springer, 2000.
2. A. T. Álvarez and J. L. F. Alemán. Formally modeling UML and its evolution: A holistic approach. In S. Smith and C. Talcott, editors, *Proceedings, Formal Methods for Open Object-based Distributed Systems, Stanford, USA*, pages 183–206. Kluwer, 2000.
3. T. Baar, B. Beckert, and P. H. Schmitt. An extension of Dynamic Logic for modelling OCL's @*pre* operator. In *Proceedings, Fourth Andrei Ershov International Conference, Perspectives of System Informatics, Novosibirsk, Russia*, LNCS. Springer, 2001.
4. Boldsoft, Rational Software Co., and IONA. Response to the UML 2.0 OCL RfP. Initial submission, August 2001.
5. R. Breu, R. Gosu, F. Huber, B. Rumpe, and W. Schwerin. Towards a precise semantics for object-oriented modeling techniques. In J. Bosch and S. Mitchell, editors, *ECOOP Workshop, Jyväskylä, Finland*, LNCS 1357, pages 205–210. Springer, 1998.
6. R. France. A problem-oriented analysis of basic UML static modeling concepts. In *Proceedings, Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Denver, USA*, volume 34 (10) of *ACM SIGPLAN notices*. ACM Press, 1999.
7. M. Gogolla and M. Richters. On constraints and queries in UML. In M. Schader and A. Korthaus, editors, *The Unified Modeling Language: Technical Aspects and Applications*, pages 109–121. Physica-Verlag, 1998.
8. A. Hamie, F. Civello, J. Howse, S. Kent, and M. Mitchell. Reflections on the Object Constraint Language. In *Post Workshop Proceedings of UML98*. Springer, 1998.
9. A. Hamie, J. Howse, and S. Kent. Interpreting the Object Constraint Language. In *Proceedings, Asia Pacific Conference in Software Engineering*. IEEE Press, July 1998.
10. H. Hußmann, M. Cerioli, G. Reggio, and F. Tort. Abstract data types and UML models. Technical Report DISI-TR-99-15, DISI – Università di Genova, Italy, 1999.
11. H. Hußmann, B. Demuth, and F. Finger. Modular architecture for a toolset supporting OCL. In A. Evans, S. Kent, and B. Selic, editors, *Proceedings, International Conference on the Unified Modeling Language (UML), York, UK*, LNCS 1939, pages 278–293. Springer, 2000.
12. U. Keller. Übersetzung von OCL-Constraints in Formeln einer Dynamischen Logik für Java. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, 2002. In German.
13. S.-K. Kim and D. Carrington. Formalizing the UML class diagram using Object-Z. In R. France and B. Rumpe, editors, *Proceedings, Unified Modeling Language, Fort Collins, USA*, LNCS 1723, pages 83–98. Springer, 1999.
14. P. Krishnan. Consistency checks for UML. In *Proceedings, Asia Pacific Software Engineering Conference (APSEC)*, pages 162–169, 2000.
15. Object Management Group, Inc., Framingham/MA, USA, www.omg.org. *OMG Unified Modeling Language Specification, Version 1.3*, June 1999.
16. G. Reggio, M. Cerioli, and E. Astesiano. An algebraic semantics of UML supporting its multiview approach. In D. Heylen, A. Nijholt, and G. Scollo, editors, *Proceedings, AMiLP 2000*, number 16 in Twente Workshop on Language Technology. University of Twente, 2000.
17. M. Richters and M. Gogolla. On formalizing the UML object constraint language OCL. In *Proceedings, Conceptual Modeling*, LNCS 1507, pages 449–464. Springer, 1998.
18. P. H. Schmitt. A model theoretic semantics of OCL. In B. Beckert, R. France, R. Hähnle, and B. Jacobs, editors, *Proceedings, IJCAR Workshop on Precise Modelling and Deduction for Object-oriented Software Development, Siena, Italy*, pages 43–57. Technical Report DII 07/01, Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Siena, 2001.
19. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, 1999.

Index of Authors