Model Checking and Theorem Proving: a Unified Framework

Sergey Berezin

24th January 2002

CMU-CS-02-100

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
Edmund Clarke, Chair
Randal Bryant
Todd Mowry
Kenneth McMillan, *Cadence Berkeley Labs*
Natarajan Shankar, *SRI International*

This research was sponsored by the Semiconductor Research Corporation (SRC) under contract no. 98-DJ-294 and no. 99-TJ-684, and by the National Science Foundation (NSF) under grant no. CCR-9217549 and CCR-9803774. 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, the U.S. government or any other entity.

Keywords: Formal methods, model checking, theorem proving, *SyMP*, temporal logics, hardware verification, Tomasulo's algorithm, security protocols, *Athena*, proof systems, SML.

Dedicated to my parents and my wife Angela

Abstract

The never-ending growth of the complexity of modern hardware and software systems requires more and more sophisticated methods of verification. The *state space explosion problem* leaves little hope for automatic finite-state verification techniques like *model checking* to remain practical, especially when designs become *parameterized*. The use of *theorem proving* techniques is inevitable to cope with the new verification challenges. "Pure" theorem proving, on the other hand, can also be quite tedious and impractical for complex designs. Ideally, one would like to find an efficient combination of model checking and theorem proving, and the quest for such a combination has long been one of the major challenges in the field of formal verification.

Many new methodologies have been proposed to make the two techniques work in ensemble. Observing such a wide variety of methodologies, one may even question the mere possibility of finding a universal technique that would combine model checking and theorem proving. Instead, it seems more practical to expand the collection of these problem-specific methodologies.

The development of new methodologies is usually an iterative experimental process in which researchers implement their ideas in a prototype tool and run several verification examples in it. The experiments provide the necessary feedback for refining the methodology and generalizing it to handle wider class of examples, or give hints on how to tune the technique to specific applications.

Since the methodologies often use both model checking and theorem proving techniques, implementing new tools becomes the main bottleneck in their development. In this work, we provide a new unified framework that includes both model checking and theorem proving, and is designed for fast prototyping of tools or manual but computer-assisted testing of new verification methodologies. The tool *SyMP* (*Symbolic Model Prover*) implements this framework in a theorem prover-like environment. Moreover, the tool is in fact a programmer's kit for generating new, possibly highly specialized, theorem provers. It provides a base for the development of new tools for emerging methodologies and reduces the implementation time. The architecture of the tool and the theory behind it help organizing the new methodologies in a systematic and extensible way.

Acknowledgments

I almost cannot believe that this long thesis marathon is nearing the end, and I am greatly indebted to many people who helped me get started, pointed in the right direction, kept me on track, and supported along the way, and without whom I would have never made it.

First of all, I would like to thank my advisor, Edmund Clarke, who provided the invaluable guidance and financial support throughout my entire stay at Carnegie Mellon. He has taught me innumerable things one needs as a researcher, from writing papers and grant proposals to choosing the right projects and succeeding in them. I am especially grateful for his persistence in convincing me to do certain parts of my work, which, as I discovered later, turned out to be absolutely necessary, and without them my work would be incomplete.

Thanks to all my committee members for taking their time to serve on the committee, and for the feedback they have provided. Most of all, I appreciate the detailed discussions with Ken McMillan about *SMV*, the theory behind it and practical issues, and many interesting subtleties in the abstraction mechanism and its implementation.

Many thanks to the members of our research group, it has been my great pleasure to work with so many bright and talented people. Alex Groce has significantly contributed to the development of my tool *SyMP*, and in particular, wrote the first version of the *SyMP*-to-*SMV* converter and all of the infrastructure for the Athena proof system, which was crucial to my finishing the implementation in time. And it is only due to Dawn Song that the Athena module has become possible, as she came up with and developed the theory of it, and later we have had extensive discussions about its implementation details. Daniel Kröning supplied us with interesting real-life hardware examples (FPU controller, Tomasulo's algorithm, etc.), has started to implement yet another extension module for *SyMP*, contributing to its versatility, and, in addition, has proof-read my thesis. And thanks to the other group members for a lot of inspiring and insightful discussions: Somesh Jha, Sérgio Campos, Yuan Lu, Pankaj Chauhan, Helmut Veith, Dong Wang, Armin Biere, Yunshan Zhu, and everyone else.

Spending two summers (1997-98) in the formal verification group at SRI International opened my eyes on many real-life aspects of theorem proving, and it played a big role in choosing my thesis topic. I would like to thank everyone who worked with me there, and most notably, N. Shankar and Sam Owre for a lot of help with the *PVS* theorem prover.

Many thanks to Steve German from IBM for his example of a cache coherence protocol, which has become one of the central examples in my work.

I would also like to thank my former undergraduate advisor Dr. Shilov for teaching me the basics of formal verification and turning me into a researcher. He deserves a big part of the credit in many of my achievements in my early career for his knowledge in the area, readiness to help, and his friendly guidance.

Furthermore, I cannot imagine surviving the tense life of a graduate student within gloomy walls of my office without my friendly officemates, Owen Cheng and Muralidar Taupur, helping me stay upbeat, letting me vent all my thesis bitterness, and assisting with many small technical problems.

I greatly appreciate the work of CMU CS help desk and SCS in general for providing a truly unique environment where everything actually works, to the extent possible and sometimes even quite a bit more. On the same note, thanks to the CS zephyr community for the invaluable source of collective wisdom.

And finally, most of my gratitude goes to my dear wife, Angela, for her unsurpassed gift of persuasiveness, great talent of ferreting out information, and amazing faith in what I always thought as impossible. Without her, I would have not even come to CMU in the first place. With her help and support, I have made it through the application process, moved to Pittsburgh, stayed sane and in working order for all six years, and finally is about to receive my degree from CMU — something I have never even dreamed of before I met her. I cannot thank her enough for being patient and caring, enduring all the hardship of being a graduate student's wife, and doing everything for me around the house that I was supposed to do instead of working on my thesis.

Contents

Chapter 1

Introduction

As hardware components become more and more complex, the task of formally verifying them also becomes increasingly difficult. The two major verification approaches, *model checking* and *theorem proving*, have long reached their limitations as general-purpose techniques, and most of the research now is concentrated on efficient specialization of both approaches to relatively narrow problem domains. There are a host of such special-purpose techniques developed both in model checking and theorem proving communities that often allow formal verification to be applicable to amazingly large and complex systems.

The never-ending growth of the complexity of modern hardware and software systems requires more and more sophisticated methods of verification. The *state space explosion problem* leaves little hope for automatic finite-state verification techniques like *model checking* to remain practical, especially when designs become *parameterized*. The use of *theorem proving* techniques is inevitable to cope with the new verification challenges. "Pure" theorem proving, on the other hand, can also be quite tedious and impractical for complex designs. Ideally, one would like to find an efficient combination of model checking and theorem proving, and the quest for such a combination has long been one of the major challenges in the field of formal verification.

Many new methodologies have been proposed to make the two techniques work in ensemble. Observing such a wide variety of methodologies, one may even question the mere possibility of finding a universal technique that would combine model checking and theorem proving. Instead, it seems more practical to expand the collection of these problem-specific methodologies.

The development of new methodologies is usually an iterative experimental process in which researchers implement their ideas in a prototype tool and run several verification examples in it. The experiments provide the necessary feedback for refining the methodology and generalizing it to handle wider class of examples, or give hints on how to tune the technique to specific applications.

Since the methodologies often use both model checking and theorem proving techniques, implementing new tools becomes the main bottleneck in the their development. In this work, we provide a new unified framework that includes both model checking and theorem proving, and is designed for fast tool prototyping or manual but computer-assisted testing of new verification methodologies. The tool *SyMP* (*Symbolic Model Prover*) implements this methodology in a theorem prover-like environment. Moreover, the tool is in fact a programmer's kit for generating new, possibly highly specialized, theorem provers. It provides a basis for the development of new tools which support emerging methodologies, and reduces the implementation time. The architecture of the tool and the theory behind it help to organize the new methodologies in a systematic and extensible way.

1.1 Verification Problem

Typically, a *formal verification problem* is a problem of *proving* that a design meets certain specifications. For instance, a design of an arbiter for a shared resource has to satisfy the *mutual exclusion property* (no two contenders for the resource have access to it at the same time), which becomes the specification.

The original design may be written in some programming language like C or a hardware description language like Verilog. For our purposes, it is not important what concrete language is used to implement the design. However, we will often assume that the encoding is done using *variables* and *assignments* to these variables.

In order to reason about a design mathematically, we formalize it in terms of a *state machine*, or a *Kripke structure*:

$$M = (S, \rightarrow, I),$$

where S is a set of *states*, $I \subseteq S$ is a set of *initial state*s, and $\rightarrow \subseteq S \times S$ is a *transition relation*. We will also refer to M as a *model*. A state $s \in S$ in the model corresponds to an assignment of specific values to all of the variables in the design, including the program counter. A transition from one state to another models the execution of one program step which updates the program counter and possibly assigns new values to some variables. The set of initial states I is derived from the variable initialization code of the design.

The property is formalized as a logical formula f, often in some temporal logic like CTL or LTL. The verification problem is now formally stated as a satisfiability problem of the formula f in the model M:

$$M \models f.$$

1.2 Existing Verification Techniques

There have been quite a number of different methodologies proposed in the past few years to help overcome the drawbacks of "pure" model checking and theorem proving. Some techniques augment model checking with new state reductions, others provide meta-reasoning to simplify the model before running a model checker on it. In theorem proving, a very ubiquitous and conceptually simple technique is to add a model checker as a decision procedure to discharge finite-state subgoals. Many other methodologies help to systematize the proofs, so they become smaller, more manageable, and even more automatic. Yet a few methodologies tackle the very problem of combining model checking and theorem proving techniques at a rather general level.

Below we briefly describe some of the most common verification methodologies. Our claim is that they can all be expressed in our new framework without any loss of generality, and more importantly, without much loss of efficiency. The latter is a very important point — after all, anything in formal verification can in principle be expressed in *Higher Order Logic* (HOL), and therefore, done in a pure theorem proving environment. But in practice, this often comes at a very high cost of losing efficiency. Therefore, keeping the efficiency and the degree of automation is an important factor in our contribution.

1.2.1 Cone of Influence Reduction

Cone of Influence [BCC97] is a very straightforward but effective state reduction technique. Before running a model checker on the model, one finds the set of variables that can potentially affect the specification, and removes all the other variables from the model. The dependency is computed by first taking the variables that directly occur in the specification, adding to this set those variables that appear on the right hand side of the assignments to the variables already in the set, and doing this repeatedly until no new variables can be added.

1.2.2 Abstraction and Symmetry Reductions

When the original model has symmetric components (e.g. an array of caches in a shared memory protocol), it is often possible to reduce the number of these components by re-ordering them dynamically [CJ95]. This way we can force the components that influence our specification to be always some particular selected components. Since the other components will not be important, we can then remove them from the model.

Alternatively, when the model is too large or even infinite, one may use *abstraction* to reduce the size of the model [Lon93]. Some constraints are removed from the original model and replaced by nondeterminism, making the model smaller (it has fewer constraints on the transitions), but with "more behaviors," so that the abstract model has more ways of reaching erroneous states. Therefore, if a safety property is true in the abstract model, then it must be true in the original model as well. These techniques are discussed a bit further in Section 2.3.1.

1.2.3 Assume-Guarantee Reasoning

Assume-Guarantee Reasoning [Lon93, Pnu85] is a variant of *compositional reasoning* when some properties are proven for each of the component, and then the property of interest is derived for their parallel composition. The properties of some components are proven under certain *assumptions* about other components, and then those components are proven to *guarantee* these assumptions.

1.2.4 Inductive Proofs

The reduction techniques mentioned above come mostly from the model checking world. We now discuss some of the verification techniques from theorem proving.

From the point of view of temporal specifications, there are two types of induction that can be applied. One is *induction on time*, and the other is *induction on the data structures*.

Safety properties in theorem proving are often proven by induction on time. First, one proves that the property holds in the initial states (the base of the induction), and then, assuming that the property holds in some arbitrary state, one proves that all the states in its transition image also satisfy this property (inductive step). Since the original property is rarely inductive (not strong enough to satisfy the inductive step), it is often necessary to *strengthen* the invariant before it can be proven, and this is usually the hardest and the least automatic step in the verification. Nowadays there are a few tools that help compute inductive invariants automatically [BLO98, GS96].

While proving properties about complex or infinite data structures, one may need to use natural or structural induction within the current state of the system.

1.2.5 "Circular" Compositional Reasoning

A combination of assume-guarantee and induction on time yields the so-called *"circular" compositional reasoning* [McM98]. In some systems with tightly coupled components, the dataflow goes back and forth among the components, and to prove the guarantees of one module, we need to assume some properties about the other, and *vice versa*. The classical assume-guarantee rule does not work in this case, since we cannot break the cyclic dependency. However, we can require that the assumptions about the other components be considered only up to time t while proving the property at time $t + 1$. The base of this induction is, as usual, to show that all the guarantees are satisfied at time $t = 0$, and then the actual assume-guarantee reasoning proves the inductive step.

1.2.6 Symbolic Simulation

Symbolic simulation [BD94, Gre98] is a method of "running" a hardware device or a software program with symbolic inputs (*terms*) instead of concrete bit values. The result is a term built from these inputs and functions that the design applies to the inputs when it executes. The concrete functions in the design are often replaced with *uninterpreted functions*, both for efficiency and generality. Interpretations of the resulting term represent possible outputs that the device may produce. The properties of interest are then proven directly on the result term, usually automatically, using decision procedures for uninterpreted functions with equality like SVC [BDL96].

1.2.7 Completion Function Approach

If the above methods are generally applicable to virtually any type of a model, the method of *completion functions* [HSG98] has been developed with the focus on pipelined and superscalar microprocessors. The idea is to prove the *Burch and Dill commutative diagram* [BD94] using special user-defined function that computes the state of the microprocessor after flushing it from the current state. When these functions are simpler than the direct flushing of the machine, the verification can be done much easier. However, the user has to provide the completion functions manually, and these functions must be proven to be equivalent to the direct flushing, which is an additional overhead of the method.

1.3 Our Contribution

Our contribution to the field of formal verification is two-fold. First, we have developed a framework for combining model checking and theorem proving in such a way that one does not dominate the other, and all of the main advantages of both techniques are preserved. The resulting system, therefore, has the same expressive power as a theorem prover, but at the same time enjoys the same degree of automation and speed as the state-of-the-art model checkers for finite-state part of verification. In addition, many powerful transformations can now be included into the system that were not directly present in any "pure" model checker or theorem prover, such as induction on time or various types of abstraction.

Second, we have extended the notion of theorem proving from its traditional role in formal verification to a more abstract one which allows us to specialize it very efficiently to a wide range of specific problem domains. In particular, the framework for combining model checking with theorem proving becomes one such specialization. We have also built a tool called *SyMP* [Ber01] (stands for *Symbolic Model Prover*) that supports this extension of theorem proving and, in effect, is a *theorem prover generator*, or a programmer's kit that simplifies the task of building new theorem provers for various specific problem domains. To date, this tool has been tested on four problem domains: combination of model checking and theorem proving for hardware verification, security protocol analysis using the *Athena* [SBP01] approach, verification of C programs in the *Reedpipe/CProver* proof system implemented by Daniel Kröning, and an emerging combination of Presburger arithmetic with a bit-vector theory based on Verilog operations. Also, the *Analytica* [CZ92] theorem prover, originally implemented in *Mathematica*, will be re-implemented as yet another proof system in *SyMP*.

1.3.1 The Framework for Combining Model Checking and Theorem Proving

A *model checking problem* is a problem of verifying that a formula f holds in a model M:

$$M \models f,$$

where M represents the design and is usually finite, and f is the desired property of this design expressed in a temporal logic like CTL or LTL. Model checking techniques are based on the exhaustive state traversal of the model, and are often automatic and very efficient. The efficiency comes from the use of compact data structures like OBDDs [Bry86]

(Ordered Binary Decision Diagrams) to represent the model, and powerful reduction techniques to prune the search space. However, model checking is largely limited to finite models and propositional formulas.

A *theorem proving problem*, on the other hand, is a problem of checking the *validity* of a formula f, that is, f must hold in *every* model. The logic is often much more expressive than propositional temporal logic, and the properties over infinite domains can be easily expressed. A theorem proving problem is solved by finding a *proof* of the formula (which is then called a *theorem*) in a *proof system*.

In order to use theorem proving, we need to translate our verification problem into a theorem proving problem. This entails encoding both the design and the (possibly temporal) specification into the logic of a theorem prover.

Since many theorem provers use higher-order logic (HOL), the expressive power of the approach is virtually unlimited. In practice, however, the required expertise and the amount of manual guidance in finding a proof may become prohibitively high.

In our framework, we attempt to strike the balance between the expressiveness of theorem proving and efficiency and automation of model checking. In our opinion, one of the major bottlenecks in theorem proving, and the reason for success in model checking, is the representation of the problem. In case of model checking, the problem corresponds very closely to the verification problem (design becomes the model, property is rewritten as a formula), while in theorem proving a non-trivial translation is required, which may destroy crucial information about the design structure. Therefore, taking the efficient representation ideas from model checking and incorporating them in theorem proving promises to yield a more powerful hybrid approach, which we call *model proving*.

To put it simply, our framework is a theorem prover based on a special-purpose proof system. Since the problem we are solving involves a complex model and a usually relatively simple property, the sequent in this proof system includes the model explicitly, and otherwise can be thought of as a Gentzen sequent. In its simple version, it can be written as follows:

$$M;\, \Gamma \Longrightarrow \Delta,$$

where M is the model, Γ is the set of *assumptions* (first-order CTL, LTL or μ-calculus formulas), and Δ is the set of *conclusions*.

The inference rules include all the rules similar to the ones in the classical Gentzen proof system for the higher-order logic, and contain additional rules that operate on the model and the modal operators of the logic. For example, model checking as a decision procedure can be introduced as a rule:

$$\frac{\mathrm{MC}(M,\, \bigwedge\Gamma \to \bigvee\Delta) = \mathbf{true}}{\mathrm{M};\, \Gamma \Longrightarrow \Delta}\,\mathrm{MC},$$

where $\mathrm{MC}(M,\, \phi)$ is a function that runs a model checking engine and reports whether M satisfies ϕ. Since the model is explicitly present in the sequent, there is no need to extract it from the formulas, as does, for instance, the PVS prover [SOR93]. Also, M is stored in the sequent in a convenient and efficient representation, and in particular, may preserve structural information that can be exploited to make the proof construction easier and more automatic. A simple example of such a use would be the *cone of influence reduction*, which again is just another rule in our proof system:

$$\frac{V = \mathrm{COI}(M,\, \Gamma \cup \Delta) \quad M|_V;\, \Gamma \Longrightarrow \Delta}{M;\, \Gamma \Longrightarrow \Delta}\,\mathrm{cone}.$$

Here $\mathrm{COI}(M,\, \Phi)$ computes the set of variables in the cone of influence of all formulas in Φ, and then the model is restricted to only those variables.

The advantages of such a system are not only in the fact that model checking and theorem proving can be used at the same prompt. The proof rules can now be constructed specifically for the transformations that one uses most often in hardware verification. Besides the model checking procedure and the cone of influence reduction (which are already atomic rules at this point), we can add various abstractions, *case splitting*, circular reasoning, and many others as atomic rules. Now, when the user outlines the verification plan for his example, he can then simply apply the appropriate rules that correspond one-to-one to the high-level steps in the outline on the paper. This means the verification itself can be done on a higher level, the proofs get simpler, and more low-level steps are performed by the tool automatically inside the high-level rules, compared to the traditional theorem proving.

The proof system that we actually use has many other powerful proof rules, a slightly more complex sequent, and it operates on full branching time and linear time μ-calculus. The rigorous definition of this proof system is in Chapter 3.

1.3.2 The Framework for Specializing Theorem Proving to Different Problem Domains

As we have already mentioned, most of the advances in formal methods are done by designing new verification methodologies for specialized problem domains. Often, these new methodologies are based on existing, "traditional" approaches and corresponding tools. In practice, however, the available techniques and their implementations often present intrinsic problems that significantly slow down the development of new verification methods. These problems stem mostly from the artificial standards imposed on researchers in both model checking and theorem proving by various factors ranging from the stereotypes in the foundations to implementation artifacts in the tools.

If we take one more look at the previous Section 1.3.1, the reason we can achieve a combination of model checking and theorem proving is quite simple in its root: we use a custom sequent and a specialized proof system for our specific problem domain, as opposed to using an "all-in-one" proof system of an off-the-shelf general purpose theorem prover. A natural question arises whether we can apply a similar approach to other problem domains than hardware verification. While designing the architecture of our tool *SyMP*, we conjectured that allowing the tool to work with arbitrary sequents and proof systems would greatly increase it versatility.

To test this conjecture, we have developed a general purpose *prover generator* kit that takes a data structure for the sequent and a proof system as a parameter, and generates an interactive theorem prover with a user interface (Emacs) and proof management (keeping track of the proof tree, editing the proof, automated proof search using strategies, etc.).

Our first proof system, of course, was the one described above for combining model checking and theorem proving, which fits very nicely in the framework. But for testing our conjecture, we needed at least one more example of some clearly different problem domain. We picked security protocol verification, since it is about as far from hardware verification as one can get in formal methods, and, moreover, it has very little to do with traditional notions of theorem proving and model checking. If this problem domain can be efficiently implemented in our tool, then most likely many other domains not as alien to theorem proving have a good chance of finding their place in *SyMP* too.

Our proof system of choice for security protocols became *Athena*, a verification methodology originally developed by Dawn Song [Son99] based on the *strand space model* [THG98b, FHG98, THG99], and later reformulated in our joint work as a specialized proof system [SBP01]. *Athena* implementation in *SyMP* was developed quite fast (about 2 person-months total of actual effort), and the result was a very efficient automatic theorem prover highly specialized to security protocols, and with an option of interactive user guidance with a convenient interface. The automatic proof search is comparable in speed and efficiency to the original implementation of *Athena* by Dawn Song up to some constant factor due to the overhead for the proof management and the interactive user interface. We will talk more about the implementation itself in Section 4.3, and the experimental results obtained using it are in Section 5.3.

We believe that the success of our implementation of *Athena* strongly supports our conjecture that theorem provers parameterized by customized proof systems can be much more versatile and efficient than conventional ones, that is, those based on a fixed and very expressive logic like HOL. The implementation of two more proof systems, *Reedpipe/CProver* for the verification of C programs and *Bitvector* that combines bit-vector theories with Presburger arithmetic, also show a lot of promise. Another proof system *Analytica* [CZ92] is also emerging, which was originally a stand-alone theorem prover implemented in *Mathematica* and specialized in elementary analysis.

The notion of a special-purpose proof system is, of course, not new and has been studied in the field of *Automated Deduction* quite extensively, resulting, in particular, in a creation of *Logical Framework* [Pfe99b, Pfe99a, PS99, Pfe94]. Logical framework (LF) can be viewed as a way to specify the sequent and the inference rules of a user-defined proof system, and letting a tool search for proofs automatically, or build a custom theorem prover for this proof system.

The difference between our approach and LF is that LF provides the user with a fixed *meta-language* for defining the proof system, and usually has limited expressive power (on purpose) to ease the automated proof search and make use of certain theoretical results that apply to such a language to increase the efficiency. The fixed and restricted meta-language is often expressive enough for various forms of the first- or higher-order logic or similar formalisms, but can be quite restrictive in many other problem domains. In fact, at least two of our proof systems (model checking + theorem proving and *Athena*) cannot be efficiently expressed in LF. In case of *Athena*, the sequent is essentially a graph of a special form representing the set of runs of a security protocol, and proof rules are rather complex transformations on this graph. Since LF is mostly targeted at proof systems where inference rules contain only relatively simple formula templates, encoding an arbitrary graph, and especially manipulating it efficiently in LF is a challenging, if at all a feasible, task.

In contrast to LF, we let the proof system developer use the full power of SML (general-purpose programming language and the implementation language of *SyMP*) to express his or her sequents and rules. The result is, perhaps, not as automatic a prover as it can be with LF, but it is much better customized to the problem domain and has an interactive mode which is invaluable for designing verification methodologies in the new fields, where the only available approach is by trial and error. Implementing a proof system in our framework might be more difficult than in LF, since it is done on a lower level, but the advantages of the resulting tool are often worth it, as we have seen from our experience in developing the four proof systems.

Chapter 2

Background and Related Work

Below we briefly describe the theory of model checking, theorem proving, and some well-known approaches to their combination. The main purpose of this chapter is to introduce the notation used throughout the document, and make the document more self-contained.

2.1 Model Checking

In general, a *model checking problem* is a problem of checking whether a given model satisfies a given property:

$$M \models \phi.$$

Normally, the model M represents a design, and the property ϕ formalizes its correctness criteria. Traditionally, model checking has focused mostly on automatic decision procedures for solving its verification problem. Therefore, to guarantee termination, the model is often restricted to a *finite state transition system*, and properties are expressed in a *propositional temporal logic* like CTL or LTL, for which finite-state model checking is known to be decidable.

It is important to understand that model checking problem is not limited to finite state systems or propositional logics. In fact, most of our results in the later chapters will deal with infinite state or parameterized model checking problems. Having said that, we are now going to define formally all of the notions mentioned above in the context of traditional finite state model checking.

2.1.1 Model: Kripke Structure

A *Transition System*, or a *Kripke Structure*, is a tuple

$$M = (S_M, \to_M, I_M, L_M),$$

where

- S_M is a non-empty set of *states*;
- $\to_M \subseteq S_M \times S_M$ is a *transition relation.* When a pair of states (s, q) belongs to $\to$ (denoted $s \to q$), we say that s *has a transition to* q;
- $I_M \subseteq S_M$ is a set of *initial states*;
- and $L_M : A \to 2^{S_M}$ is a *labeling function* which labels states with *atomic propositions* from the set A. We postpone the discussion of the labeling function to the next subsection.

We often omit L_M in the model when its definition is clear from the context. For instance, when atomic propositions are defined using predicates (equalities, arithmetic comparisons, etc.) on *state variables*, the labeling function L_M is implicitly derived from the semantics of these predicates.

In the sequel, we will often use the terms *model*, *transition system*, and *Kripke structure* interchangeably.

Definition 2.1.1. Transition relation $\to_M$ in a model M is called *total* if for every state $s \in S_M$ there is (possibly the same) $q \in S_M$ such that $s \to_M q$. That is, every state in the model has an outgoing transition.

Later, in the definitions of CTL and LTL we always assume that the transition relation is total. The semantics for these logics will be undefined for models with non-total transition relations.

Definition 2.1.2. A *path* in M is a finite or infinite sequence of states $\pi = s_0, s_1, \ldots$ such that for every $i \geq 0$, $s_i \to s_{i+1}$.

We denote individual states of π by their subindices: for instance, π_0 is the first state of the path. A superindexed path π^i denotes a suffix of π that starts at π_i. In particular, $\pi^0 = \pi$.

Definition 2.1.3. A state s is called *reachable* in M, if there exists a finite path π in M such that $\pi_0 \in I_M$, and the last state of π is s. In other words, there exists a path from some initial state to the state s in M.

In practice, transition systems are often described using special input languages that resemble conventional programming languages (e.g. SMV [McM93]). The values of program variables $\vec{x} = (x_1, \ldots, x_n)$ determine the current state of the system: $s = (v_1, \ldots, v_n)$. Hence, the set of states S is a Cartesian product of the variables' ranges, or *types*: $S = \tau_1 \times \cdots \times \tau_n$. The set of initial states is defined by assigning initial values to some of the variables:

init(x_i) := v_i,

and the transition relation is represented by the *next state* assignments of the form

next(x_i) := $f_i(\vec{x})$.

All such assignments are "executed" concurrently, and this way the system transitions to the next global state.

One of the advantages of such a language is that it is straightforward to generate a propositional formula $\mathcal{T}(\vec{x}, \vec{x}')$ representing the transition relation:

$$\mathcal{T}(\vec{x}, \vec{x}') = \bigwedge_{i=1}^{n} (x_i' = f_i(\vec{x})).$$

The actual transition relation is the set of pairs of states satisfying this formula. The same holds for the set of initial states.

2.1.2 Property: Temporal Logic

The properties of transition systems are expressed in *temporal logics*, most often in propositional CTL or LTL, and sometimes in μ-calculus. We first introduce the logics CTL and LTL, then define branching and linear time versions of μ-calculus, and show how CTL and LTL can be expressed in it. Later, we will use the appropriate version of μ-calculus for theoretical definitions and theorems, and CTL and LTL as a syntactic sugar in the actual specifications for real systems.

In all of the logics, the atomic formulas are the *atomic propositions* from A. Each state s in a model M has a set of atomic propositions $L(s)$ that are true in this state, where L is the labeling function of M; all the other atomic propositions are false in s. For example, in an SMV-like language an atomic proposition $x = y$ will belong to $L(s)$ for those states s where the state variables x and y are assigned the same values.

All logics also have a common set of *propositional connectives*:

$$\neg\phi,\ \phi_1 \wedge \phi_2,\ \phi_1 \vee \phi_2,\ \phi_1 \rightarrow \phi_2.$$

Here and below ϕ, ϕ_1, and ϕ_2 are always assumed to be formulas from the appropriate logic. Below we define only those constructors that are different for different logics, namely, the temporal and fixpoint operators.

The formulas of all temporal logics are interpreted over sets of states where they are true. For each logic, we define the *semantic function*

$$[\![\cdot]\!] : \mathrm{Model} \times \mathrm{Formula}\,[\times(\mathrm{Var} \rightarrow 2^S)\,]\ \rightarrow 2^S$$

that assigns to each formula ϕ the sets of states $[\![\phi]\!]_M e$ where it is true. This function will have an additional *environment* argument $e : \mathrm{Var} \rightarrow 2^S$ for μ-calculus, which assigns interpretations to *propositional variables*. Since CTL and LTL do not have propositional variables, their semantics does not depend on e, and we omit it from the definition. We will also often omit the model from the subindex when the model is unambiguously understood from the context.

When we need to say that a formula ϕ holds in a particular state s, instead of writing $s \in [\![\phi]\!]_M$ we sometimes will use another common notation:

$$M,\ s \models \phi.$$

Since including the environment in this notation is a bit cumbersome, we use it only for CTL, LTL, and closed μ-calculus formulas (those that do not contain free propositional variables). A variant of this notation is

$$M \models \phi,$$

which means that ϕ holds in *all the initial states of* M. In this case, we say that ϕ holds in the model M.

Linear Time Temporal Logic: LTL.

$\mathbf{X}\,\phi$	Next time operator
$\mathbf{G}\,\phi$	"Always true" operator (or "**G**lobally")
$\mathbf{F}\,\phi$	"Eventually true" operator (or "in the **F**uture")
$\phi_1\,\mathbf{U}\,\phi_2$	"**U**ntil" operator
$\phi_1\,\mathbf{R}\,\phi_2$	"**R**elease" operator

The validity of formulas in LTL is originally defined on a path rather than on an individual state, and we say that an LTL formula ϕ holds in a state s iff ϕ holds on *all paths* π starting from s (denoted as $M, \pi \models \phi$, or just $\pi \models \phi$, when M is understood from the context). We only define the semantics of LTL formulas for models with total transition relation, so every path is essentially an infinite path. Formally,

$$[\![\phi]\!]_M = \{s \mid \forall \pi.\, \pi_0 = s \to M, \pi \models \phi\}.$$

The relation $M, \pi \models \phi$ defines the semantics of LTL:

$$\begin{array}{rcl}
\text{if } \phi \in A, \text{then } M, \pi \models \phi & \text{iff} & \pi_0 \in L_M(\phi) \\
M, \pi \models X\,\phi & \text{iff} & M, \pi^1 \models \phi \\
M, \pi \models \mathbf{G}\,\phi & \text{iff} & \forall i \geq 0.\, M, \pi^i \models \phi \\
M, \pi \models \mathbf{F}\,\phi & \text{iff} & \exists i \geq 0.\, M, \pi^i \models \phi \\
M, \pi \models \phi_1\,\mathbf{U}\,\phi_2 & \text{iff} & \exists i \geq 0.\, M, \pi^i \models \phi_2 \text{ and } \forall 0 \leq j < i.\, M, \pi^j \models \phi_1 \\
M, \pi \models \phi_1\,\mathbf{R}\,\phi_2 & \text{iff} & (\forall k \geq 0.\, M, \pi^k \models \phi_2) \\
 & & \bigvee(\exists i \geq 0.\, M, \pi^i \models \phi_1 \text{ and } \forall 0 \leq j \leq i.\, M, \pi^j \models \phi_2).
\end{array}$$

Intuitively, $\mathbf{X}\,\phi$ means that on a given path π, the formula ϕ must hold in the second state of this path π_1 (remember that the first state is π_0); $\mathbf{G}\,\phi$ means that ϕ holds everywhere along the path π, or more formally, on every suffix of π. The "until" operator $\phi_1\,\mathbf{U}\,\phi_2$ means that there is a point along π where ϕ_2 holds, and everywhere before that ϕ_1 must hold. It is important that ϕ_2 must eventually hold, otherwise the entire *until* operator does not hold. After the point where ϕ_2 holds for the first time, the validity of both subformulas does not influence the validity of the entire formula. The dual of the "until", the "release" operator $\phi_1\,\mathbf{R}\,\phi_2$ means that ϕ_2 must hold along the path until and including the moment when ϕ_1 becomes true; in other words, ϕ_1 *releases* ϕ_2 from having to hold on the rest of the path. Notice, that unlike in the "until," ϕ_1 may not hold anywhere on the path, in which case ϕ_2 will have to remain true forever.

Branching Time Temporal Logic: CTL.

$\mathbf{AX}\,\phi$, $\mathbf{EX}\,\phi$	Next time operators
$\mathbf{AG}\,\phi$, $\mathbf{EG}\,\phi$	"Always true" operators (or "**G**lobally")
$\mathbf{AF}\,\phi$, $\mathbf{EF}\,\phi$	"Eventually true" operators (or "in the **F**uture")
$\mathbf{A}[\phi_1\,\mathbf{U}\,\phi_2]$, $\mathbf{E}[\phi_1\,\mathbf{U}\,\phi_2]$	"**U**ntil" operators
$\mathbf{A}[\phi_1\,\mathbf{R}\,\phi_2]$, $\mathbf{E}[\phi_1\,\mathbf{R}\,\phi_2]$	"**R**elease" operators

The first letter in the CTL operators denotes a *path quantifier*, and is understood as *for all paths* (**A**) or *there exists a path* (**E**). The second letter is an operator on the path with the same meaning as in LTL. The interpretation of CTL formulas is defined directly on the set of states, but we also use the notion of a path. Similarly to LTL, we only define the semantics for the models with total transition relation.

$$\begin{array}{rcl}
\text{if } \phi \in A, \text{ then } M,\, s \models \phi & \text{iff} & s \in L_M(\phi) \\
M,\, s \models \mathbf{AX}\,\phi & \text{iff} & \forall s'.\, s \to_M s' \text{ implies } M,\, s' \models \phi \\
M,\, s \models \mathbf{EX}\,\phi & \text{iff} & \exists s'.\, s \to_M s' \text{ and } M,\, s' \models \phi \\
M,\, s \models \mathbf{AG}\,\phi & \text{iff} & \forall \pi.\, \forall i \geq 0.\, \pi_0 = s \text{ implies } M,\, \pi_i \models \phi \\
M,\, s \models \mathbf{EG}\,\phi & \text{iff} & \exists \pi.\, \forall i \geq 0.\, \pi_0 = s \text{ and } M,\, \pi_i \models \phi \\
M,\, s \models \mathbf{AF}\,\phi & \text{iff} & \forall \pi.\, \exists i \geq 0.\, \pi_0 = s \text{ implies } M,\, \pi_i \models \phi \\
M,\, s \models \mathbf{EF}\,\phi & \text{iff} & \exists \pi.\, \exists i \geq 0.\, \pi_0 = s \text{ and } M,\, \pi_i \models \phi \\
M,\, s \models \mathbf{A}[\phi_1\,\mathbf{U}\,\phi_2] & \text{iff} & \forall \pi.\, \pi_0 = s \text{ implies } \exists i \geq 0.\, M,\, \pi_i \models \phi_2 \text{ and } \forall j < i.\, M,\, \pi_j \models \phi_1 \\
M,\, s \models \mathbf{E}[\phi_1\,\mathbf{U}\,\phi_2] & \text{iff} & \exists \pi.\, \pi_0 = s \text{ and } \exists i \geq 0.\, M,\, \pi_i \models \phi_2 \text{ and } \forall j < i.\, M,\, \pi_j \models \phi_1 \\
M,\, s \models \mathbf{A}[\phi_1\,\mathbf{R}\,\phi_2] & \text{iff} & \forall \pi.\, \pi_0 = s \text{ implies } (\forall k.\, M,\, \pi^k \models \phi_2) \\
 & & \bigvee(\exists i \geq 0.\, M,\, \pi_i \models \phi_1 \text{ and } \forall j \leq i.\, M,\, \pi_j \models \phi_2) \\
M,\, s \models \mathbf{E}[\phi_1\,\mathbf{R}\,\phi_2] & \text{iff} & \exists \pi.\, \pi_0 = s \text{ and } (\forall k.\, M,\, \pi^k \models \phi_2) \\
 & & \bigvee(\exists i \geq 0.\, M,\, \pi_i \models \phi_1 \text{ and } \forall j \leq i.\, M,\, \pi_j \models \phi_2)
\end{array}$$

Branching Time μ-calculus.

$\Box\phi$, $\Diamond\phi$	Next time operators
$\mu X.\, \phi$, $\nu X.\, \phi$	Fixpoint operators; ϕ is *positive* w.r.t. X

The next time operators are exactly the same as the ones in CTL: $\Box\phi$ corresponds to $\mathbf{AX}\,\phi$, and $\Diamond\phi$ — to $\mathbf{EX}\,\phi$. The fixpoint operators define the least and the greatest fixpoints (μ and ν respectively) of the *predicate transformer* induced by ϕ over X. There is additional syntactic restriction on fixpoints: the formula under the μ or ν operator must be *positive* w.r.t. X, that is, X must appear in the scope of even number of negations in ϕ. We now define the semantics formally, and then give more intuition about all the notions in this paragraph. The semantics is defined on any model, so the transition relation does not have to be total. This time, we use the set notation for the semantics with the extra environment parameter $[\![\cdot]\!]_M e$:

$$\begin{array}{rcl}
\text{if } \phi \in A, \text{ then } [\![\phi]\!]_M e & = & L(\phi) \\
\text{for } \phi = X \text{ (prop. variable)}, [\![\phi]\!]_M e & = & e(\phi) \\
[\![\Box\phi]\!]_M e & = & \{s \mid \forall s'.\, s \to_M s' \text{ implies } s' \in [\![\phi]\!]_M e\} \\
[\![\Diamond\phi]\!]_M e & = & \{s \mid \exists s'.\, s \to_M s' \text{ and } s' \in [\![\phi]\!]_M e\} \\
[\![\mu X.\, \phi]\!]_M e & = & \bigcap\{Q \mid Q \supseteq [\![\phi]\!]_M e[X \leftarrow Q]\} \\
[\![\nu X.\, \phi]\!]_M e & = & \bigcup\{Q \mid Q \subseteq [\![\phi]\!]_M e[X \leftarrow Q]\}
\end{array}$$

The predicate transformer $\tau_\phi : 2^S \to 2^S$ mapping sets of states to sets of states is defined as follows:

$$\tau_\phi(Q) = [\![\phi]\!]_M e[X \leftarrow Q],$$

where

$$e[X \leftarrow Q](Y) = \begin{cases} Q & \text{if } X = Y \\ e(Y) & \text{otherwise.} \end{cases}$$

A fixpoint of a predicate transformer τ is a set of states Q such that $\tau(Q) = Q$. The *least fixpoint* Q_μ is a fixpoint such that for any other fixpoint Q, $Q_\mu \subseteq Q$. Similarly, the *greatest fixpoint* Q_ν is a fixpoint such that for any other fixpoint Q, $Q_\nu \supseteq Q$. If τ is *monotone* w.r.t. set inclusion, that is, $\tau(Q_1) \subseteq \tau(Q_2)$ whenever $Q_1 \subseteq Q_2$, then by Tarski's theorem [Tar55] there exist the least and the greatest fixpoints of τ, and they are equal to the following sets:

$$Q_\mu = \bigcap\{Q \mid \tau(Q) \subseteq Q\},$$

$$Q_\nu = \bigcup\{Q \mid Q \subseteq \tau(Q)\}.$$

It is not difficult to show that when ϕ is positive in X, the induced predicate transformer is monotone, therefore, both fixpoints always exist for any well-formed μ-calculus formula.

When the transition relation is total, CTL operators can be expressed in branching time μ-calculus as follows:

$$\begin{array}{rcl}
\mathbf{AG}\,\phi & = & \nu X.\,\phi \wedge \Box X \\
\mathbf{EG}\,\phi & = & \nu X.\,\phi \wedge \Diamond X \\
\mathbf{AF}\,\phi & = & \mu X.\,\phi \vee \Box X \\
\mathbf{EF}\,\phi & = & \mu X.\,\phi \vee \Diamond X \\
\mathbf{A}[\phi_1\,\mathbf{U}\,\phi_2] & = & \mu X.\,\phi_2 \vee (\phi_1 \wedge \Box X) \\
\mathbf{E}[\phi_1\,\mathbf{U}\,\phi_2] & = & \mu X.\,\phi_2 \vee (\phi_1 \wedge \Diamond X) \\
\mathbf{A}[\phi_1\,\mathbf{R}\,\phi_2] & = & \nu X.\,\phi_2 \wedge (\phi_1 \vee \Box X) \\
\mathbf{E}[\phi_1\,\mathbf{R}\,\phi_2] & = & \nu X.\,\phi_2 \wedge (\phi_1 \vee \Diamond X)
\end{array}$$

We leave out the proof of these equalities as they are well-known in the literature. The consequences of being able to express CTL in the branching time μ-calculus are that any theoretical result obtained for the μ-calculus will be automatically valid for CTL. Since μ-calculus is more expressive than CTL and has a simpler syntax, we will use it for most of our theoretical results, and then use those results for CTL without further explanations.

Furthermore, we exploit the *duality* of the temporal and fixpoint operators, also widely known in the formal verification area:

$$\begin{array}{rcl}
\mu X.\,\phi(X) & = & \neg\nu X.\,\neg\phi(\neg X)\\
\nu X.\,\phi(X) & = & \neg\mu X.\,\neg\phi(\neg X)\\
\Box\phi & = & \neg\Diamond\neg\phi\\
\Diamond\phi & = & \neg\Box\neg\phi.
\end{array}$$

The linear time μ-calculus is almost never used in model checking, and we postpone its definition till Chapter 3, where we also introduce a more expressive version of μ-calculus and show how LTL can be expressed in it.

2.1.3 Algorithm: Iterative Fixpoint Computation

Most approaches to finite-state model checking hinge on the simple but efficient algorithm for computing fixpoints of predicate transformers, and therefore, are best suited for branching time temporal logics. As we have seen in the previous section, we only need to compute the least and the greatest fixpoints to evaluate the corresponding operators in the branching time μ-calculus; the algorithm for evaluating CTL formulas can be derived from those. If fact, due to the duality between the fixpoints, we only need to provide an algorithm for computing one of them, and the algorithm for the other one can be derived from that. For the purpose if this section, we assume that the model M is finite; that is, set of states S, and hence, $\rightarrow$ and I, are finite sets.

Given a monotone predicate transformer τ, we now show how to compute the least fixpoint Q_μ for it. The algorithm computes a series of *approximations* $Q_0, Q_1, \ldots, Q_n$ until $Q_n = Q_{n+1}$, and we will show that $Q_n = Q_\mu$:

$$\begin{array}{rcl}
Q_0 & = & \emptyset\\
Q_1 & = & \tau(Q_0)\\
\vdots & & \vdots\\
Q_n & = & \tau(Q_{n-1}).
\end{array}$$

First of all, let us show that the algorithm terminates. Since $\emptyset \subseteq \tau(\emptyset)$, and hence, $Q_0 \subseteq Q_1$, by monotonicity of τ we have that $\tau(Q_0) \subseteq \tau(Q_1)$, that is, $Q_1 \subseteq Q_2$. Assuming that $Q_i \subseteq Q_{i+1}$ holds for some i, using the same argument we can show that $Q_{i+1} \subseteq Q_{i+2}$, and by induction, $Q_i \subseteq Q_{i+1}$ for any i. In other words, the series of Q_i's is a non-decreasing sequence of sets. Since the set of all states S is finite, we cannot have an infinite strictly increasing sequence of Q_i's, and the algorithm will terminate after at most $n = |S|$ steps.

Since $Q_n = Q_{n+1} = \tau(Q_n)$, the set Q_n is a fixpoint of τ, and we only need to show that it is the least fixpoint.

Suppose there exists another fixpoint Q_f of τ. Since $Q_0 = \emptyset$, we have $Q_0 \subseteq Q_f$. By monotonicity, $\tau(Q_0) \subseteq \tau(Q_f)$, and since Q_f is a fixpoint, we then have $Q_1 \subseteq Q_f$. Repeating this argument enough times (that is, by induction on the length of the sequence), we have that $Q_n \subseteq Q_f$, and since Q_f was an arbitrary fixpoint of τ, we have proven that Q_n is the least fixpoint.

The algorithm for computing the greatest fixpoint can be either derived from the duality relation, or obtained from the algorithm above by starting the iteration from the set of all states S instead of the empty set. The proof of its termination and correctness is exactly the same as for the least fixpoint.

2.1.4 Efficient Representations: OBDD and SAT

The biggest limitation of finite state model checking has always been the *state explosion problem*. If the model M is represented *explicitly* as a transition graph, then the size of the model is limited to the number of states that can be stored in the computer memory, which is a few million states with the current technology. This means that one cannot reliably apply this technique to designs with more than 20-30 binary state variables. To increase the size of the model, more efficient state representations can be used such as boolean formulas. Model checking is then done by manipulating these formulas using *Ordered Binary Decision Diagrams* (OBDDs, or just BDDs) or SAT solving techniques.

We first show how to represent a finite model M using boolean formulas, describe the representation of boolean formulas by BDDs, and then discuss SAT procedures that work directly on the boolean formulas.

Boolean Formulas.

First, observe that the iterative algorithm in Section 2.1.3 never refers to individual states of M, but only to sets of states. Any set of states Q can be characterized by a *predicate* on states: $P_Q : S \to \mathcal{B}$, where $\mathcal{B} = \{\mathbf{true}, \mathbf{false}\}$, so that $Q = \{s \mid P_Q(s)\}$. If we enumerate all the states in S in binary and assign the corresponding bit vector representation $(x_1, \ldots, x_n)$ to each of the states in S, we can then represent P_Q as a *boolean formula* that depends on the boolean variables x_i. For example, if we have only 4 states in S, we can represent them using 2-bit vectors: $(0, 0)$, $(0, 1)$, $(1, 0)$, and $(1, 1)$. A predicate for a set $Q = \{(0, 1), (1, 1)\}$ is $P_Q(x_1, x_2) \equiv x_2$.

A transition relation, as any other binary relation, is a set of *pairs* of states, and therefore, can also be written as a predicate on two states: $T(s,\, s')$. When boolean vectors are used for states, T can be characterized by a boolean predicate $T(x_1,\, \ldots,\, x_n,\, x'_1,\, \ldots,\, x'_n)$. As usual, if there is a transition from s to s', then $T(s,\, s') = \mathbf{true}$. For instance, a transition relation for our little 4-state model could be

$$T(x_1,\, x_2,\, x'_1,\, x'_2) \equiv (x_2 \wedge (x'_1 \wedge \neg x'_2)) \vee (\neg x_2 \wedge \neg x'_1).$$

That is, from both states $(0,1)$ and $(1,1)$ there is a transition to $(1,0)$, and from the other two states there are transitions to $(0,0)$ and $(0,1)$.

Given the predicates for sets of states P, P_1 and P_2, it is easy to construct predicates for their complement, union, and intersection: $\neg P$, $P_1 \vee P_2$, and $P_1 \wedge P_2$. A bit harder problem is to construct a *preimage* img_P^{-1} of a set given by P w.r.t. the transition relation T:

$$\mathrm{img}_P^{-1}(x_1,\, \ldots,\, x_n) \equiv \exists x'_1,\, \ldots,\, x'_n.\, T(x_1,\, \ldots,\, x_n,\, x'_1,\, \ldots,\, x'_n) \wedge P(x'_1,\, \ldots,\, x'_n).$$

The preimage is needed to compute the set of states for the next state operator $\diamond\phi$. The interpretation of $\Box\phi$ can be computed using the duality rule with $\diamond$. Although this formula is not directly propositional, it can be transformed into a propositional formula by expanding the quantifiers as follows: $\exists x.\, f(x) \equiv f(\mathbf{true}) \vee f(\mathbf{false})$.

OBDDs.

In the fixpoint algorithm, we must be able to compare two sets of states for equality; in other words, we need a way to tell whether two boolean formulas are exactly equivalent or not. One way to achieve this is to use SAT procedures, and we will talk about it later. But there is a more efficient way to do this if we use a *canonical representation* for boolean formulas: *Ordered Binary Decision Diagrams*, first discovered by Bryant [Bry86], and then applied to model checking by McMillan [McM93].

The easiest way to think about a BDD is as a finite automaton A over the binary alphabet $\{\mathbf{true},\, \mathbf{false}\}$, and the language of this automaton is a set of n-bit strings: $L(A) \subseteq \mathcal{B}^n$. Such a language represents a set of states Q, where each state corresponds to its boolean vector. It is well-known that any finite automaton can be reduced to a *unique minimal* automaton, and operations like complement, union and intersection can be performed directly on finite automata. Since our automata will always be acyclic, and the order of bits will always be the same, we exploit this fact to make the representation even more compact: if both transitions from an automaton state q labeled $\mathbf{true}$ and $\mathbf{false}$ lead to exactly the same new state q', then q can be removed and all its incoming transitions redirected to q'. Of course, we need to be careful to remember that one bit is omitted when transitioning to q', and we associate the automaton states with the boolean variable name x_i from the Kripke structure's state encoding that must be accepted next. Now, if we transition from q_1 marked by x_i to q_2 marked by x_j, and $j > i + 1$, then we skip $j - i - 1$ bits in the input string as if the automaton has read them and not rejected the string so far. This additional reduction does not destroy the canonicity property of the minimal automaton.

Such a minimal automaton with the additional reduction is called an *Ordered Binary Decision Diagram*. Because of the canonicity property, it is easy to test two BDDs for equality, and all the necessary set operations can be performed directly on BDDs, including the preimage computation, which is basically done by a variant of quantifier expansion.

SAT decision procedures.

Although BDDs have been extremely successful in pushing model checking capabilities as far as to handle small industrial hardware designs, they still suffer from the *state explosion problem* when the number of state bits becomes larger than a few hundreds. Since BDDs is a canonical representation of boolean formulas expressive enough to solve the boolean satisfiability problem (which is NP-complete), the size of BDDs must inevitably be exponential, assuming P$\neq$ NP. In fact, it is provably exponential for some functions, for instance, the middle bit of the output of an n-bit multiplier.

Unlike BDDs, the direct representation of the model as boolean formulas does not suffer from the space explosion, but is not canonical and requires additional efforts to check for equivalence of formulas. Modern state-of-the-art SAT solvers like **Prover** [SS90], **Grasp** [SS96], **SATO** [ZS94], **Chaff** [MMZ$^+$01], and many others can handle formulas with thousands, or even tens of thousands of boolean variables, and for certain classes of problems greatly outperform BDDs.

2.1.5 State Reduction Techniques

Despite several breakthroughs in the efficient state representation, direct model checking has not been able to go far beyond several hundreds, or in some cases, thousands bits of state variables. Clearly, this is still very far from the size of real industrial designs with millions of gates. To be able to apply model checking to larger designs, *state reduction techniques* are used that exploit some features of the model, the properties, or the problem domain to reduce the state space to a tractable size. Examples include various BDD tricks like *partitioned transition relation*, *dynamic variable reordering*, general techniques like *cone* *of influence reduction*, *symmetry*, *abstraction*, problem-specific techniques, e.g. when the original design is rewritten in a simpler way, omitting the irrelevant details, but preserving the important behavior for the property being verified.

2.2 Theorem Proving

Unlike in model checking, theorem proving solves the general validity of a formula, or a problem of whether a formula F holds in *all* models:

$$\models F.$$

Theorem proving utilizes the *proof inference* technique in some *proof system* for solving this problem. First, the problem itself is transformed into a *sequent*, a working representation for the theorem proving problem. The simplest sequent used in *natural deduction* is just $\vdash F$, corresponding directly to the problem we are solving; but in general it can be more complex. We say that a sequent *holds* when it satisfies its intended semantics. For example, $\vdash F$ is derivable in natural deduction only if the formula F holds in any model.

A proof system is collection of *inference rules* of the form:

$$\frac{P_1 \quad \cdots \quad P_n}{C}\mathrm{name},$$

where C is a conclusion sequent, and P_i's are premisses sequents. The meaning of an inference rule is, if all the premisses are derivable, then the conclusion is guaranteed to hold. This should not be confused with implication, where we assume some formulas and prove some other formulas true under those assumptions. In the case of the inference rule, the sequents in premisses must actually hold (that is, be provable), not be assumed to hold.

Some inference rules may have no premisses, in which case their conclusion automatically holds. Such rules are also called *axioms*, and they are the only means to complete the proof derivation.

A *proof* of a sequent is a *derivation tree* (or a *proof tree*) whose nodes are sequents, the root being the sequent to be proven (also called a *theorem*), and for each sequent in the tree, all of its children are premisses of some inference rule in which that sequent is a conclusion. A proof is *complete* when each sequent in the derivation tree has an associated inference rule. Note, that the leaves of the tree must have associated axioms, or inference rules without premisses.

There are two ways of building a derivation tree: bottom-up, and a "direct", or a top down method. In the direct method, the derivation tree is built from the axioms down to the original theorem. Since it is often hard to tell what axioms and rules need to be used, this approach is usually used in automated provers. The bottom up method starts with the theorem as a current subgoal, and then inference rules are applied to the current subgoals, generating new subgoals (that is, the conclusion of a rule is matched with a current subgoal, and the corresponding premisses become children of the subgoal in the derivation tree, and new current subgoals). This approach is more intuitive to the user, and is often preferred in interactive provers. Below we will only use the bottom up method for building the derivation trees.

A proof system is *sound* when the fact that all premisses are valid indeed (semantically) guarantees that the conclusion holds. Soundness is an essential property of a proof system, and it is clear that without this property a complete proof cannot guarantee that the "proven" theorem is actually true. Soundness is often proven for each rule separately, but in some cases it requires the knowledge of all the other inference rules.

A proof system is called *complete* when any valid sequent has a proof in the proof system. This property is always a global property of the proof system, and is usually important only for decidable or semi-decidable logics. When the problem becomes undecidable, completeness, if can be achieved at all, is often only of academic interest. For some proof systems, it may be even hard to formulate a meaningful criteria for completeness.

An inference rule is called *invertible* when all the premisses hold if and only if the conclusion holds. Invertibility is a useful property to have for automatic proof search. Since normally premisses are simpler than the conclusion, it often makes sense to apply all the invertible rules as many times as they can be applied, transforming the original theorem into a set of simpler subgoals, which all together are equivalent to the original theorem. In particular, if one of such subgoals is proven false, then the original theorem is guaranteed to be false. This can be an invaluable feature of the proof system for providing counterexamples.

2.2.1 Higher-Order Logic

Traditionally, the logic used in theorem proving is the classical (or sometimes intuitionistic) First- or Higher-Order logic (FOL and HOL respectively). Some other kinds of logics are also used, but since all of them can be expressed in the higher-order logic, the latter is used much more often as a general property language.

Syntax.

$P(t_1, \ldots, t_n)$	Atomic formulas: *predicates* over *terms*
$\neg\phi$, $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, $\phi_1 \rightarrow \phi_2$, etc.	Propositional connectives
$\forall x.\, \phi$, $\exists x.\, \phi$	Quantifiers over (object or predicate) variables

The *terms* in atomic formulas are defined as follows:

c, x	Constants, object variables
$f(t_1, \ldots, t_n)$	n-ary functions applied to terms

Terms are interpreted over some *object domain* $\mathcal{D}$ (that is, each term has a value in $\mathcal{D}$). Constants can be considered as 0-ary functions, ones that do not take any arguments.

In the first-order logic, the quantified variables must always be object variables; that is, in $\forall x.\, \phi$, the variable x can only be used as a term in ϕ. The higher-order logic allows quantified variables to be used as functions and predicates, and also allows predicates to have functions and other predicates as arguments.

In order to avoid paradoxes (like $\exists x.\, \neg x(x)$, "there exists a set which does not belong to itself"), all terms in the higher-order logic are often required to be *well-typed*. The simplest type system would be a lattice of sets with the minimum element $\mathcal{D}$, and the other types constructed as Cartesian products and functional types over other types. For instance, a type of a binary function $f(x_1, x_2)$ over object terms is $f : \mathcal{D} \times \mathcal{D} \rightarrow \mathcal{D}$, and a binary predicate over this function and another object term has a type $P : 2^{(\mathcal{D}\times\mathcal{D}\rightarrow\mathcal{D})\times\mathcal{D}}$. We write $\tau(t)$, or $t : \tau$, to denote the type of a term t.

Semantics.

The model $M = (\mathcal{D}, I)$ for a first- or higher-order logic formula consists of an interpretation I of all the free symbols (those that are not bound by the quantifiers). The object variables and constant symbols are interpreted as values from the object domain $\mathcal{D}$ (or values of the appropriate types for the HOL), predicate symbols are interpreted as sets of tuples of the values of their arguments, and function symbols are interpreted as functions over the interpretations of their arguments. Similarly to the μ-calculus in the model checking section 2.1.2, we will use an additional *environment* e for interpreting free variables in the formulas.

$$
\begin{array}{rcl}
[\![c]\!]_M e & = & I_M(c) \\
[\![x]\!]_M e & = & e(x) \\
[\![f(t_1, \ldots, t_n)]\!]_M e & = & I(f)([\![t_1]\!]_M e, \ldots, [\![t_n]\!]_M e) \\
[\![P(t_1, \ldots, t_n)]\!]_M e & = & ([\![t_1]\!]_M e, \ldots, [\![t_n]\!]_M e) \in I(P) \\
[\![\neg\phi]\!]_M e & = & [\![\phi]\!]_M e \text{ is false} \\
[\![\phi_1 \wedge \phi_2]\!]_M e & = & [\![\phi_1]\!]_M e \text{ is true and } [\![\phi_2]\!]_M e \text{ is true} \\
[\![\phi_1 \vee \phi_2]\!]_M e & = & [\![\phi_1]\!]_M e \text{ is true or } [\![\phi_2]\!]_M e \text{ is true} \\
[\![\forall x.\, \phi]\!]_M e & = & [\![\phi]\!]_M e[x \leftarrow d] \text{ is true for all values of } d \in \tau(x) \\
[\![\exists x.\, \phi]\!]_M e & = & [\![\phi]\!]_M e[x \leftarrow d] \text{ is true for some value } d \in \tau(x)
\end{array}
$$

We say that a formula ϕ is true (or *holds*) in a model M and the environment e, when $[\![\phi]\!]_M e$ is true. We denote this fact by

$$M \models_e \phi.$$

The formula is said to be *valid* if it holds in every model and every environment, and we denote it by

$$\models \phi.$$

Proving that a formula is valid is the main problem of (traditional) theorem proving. Since the number of models is infinite, and even each model can be infinite, direct enumeration like in model checking is completely infeasible in theorem proving. Therefore, we transform the problem of a formula validity into the problem of finding a *proof*, which is a complete derivation tree in an appropriate proof system.

2.2.2 Gentzen Proof System

One of the most widely used and the most intuitive proof systems for interactive theorem proving is the *Gentzen proof system*. The sequent used in this system consists of two sets of formulas: assumptions Γ and conclusions Δ:

$$\Gamma \Rightarrow \Delta.$$

The sequent *holds* when in every model M, if all the formulas from Γ are true, then at least one formula in Δ is true. In other words, it is equivalent to the formula $\bigwedge \Gamma \to \bigvee \Delta$ being valid.

For the convenience of notation, we write Γ, ϕ to mean $\Gamma \cup \{\phi\}$ (and similarly for Δ), and will not assume any order on formulas in the sequents. The way to read such notation should be "there is a formula ϕ in the set of assumptions (conclusions)."

The proof system consists of inference rules that transform formulas *on the left* and *on the right* (that is, in the assumptions and conclusions respectively). As we have already mentioned above, we will always build a proof tree in the bottom up manner, and say that a rule *applies* to a sequent when the sequent matches the conclusion of the rule, which produces a set of new sequents (subgoals) that must be proven in order to complete the proof.

The soundness of the proof system can be established rule-by-rule, and we argue (but not prove formally) about the soundness of some inference rules as we introduce them.

In this proof system, we use only three axioms:

$$\frac{}{\Gamma,\, A \Rightarrow \Delta,\, A} \qquad \frac{}{\Gamma \Rightarrow \Delta,\, \mathtt{true}} \qquad \frac{}{\Gamma,\, \mathbf{false} \Rightarrow \Delta}.$$

The soundness of these axioms is quite easy to establish from the definition of the sequent validity. The first axiom states that in any model, if A holds, then A holds. The second and third axioms assign the meaning to the 0-ary predicates $\mathtt{true}$ and $\mathbf{false}$, which are to hold in any model and to hold in no model respectively.

When $\Gamma = \emptyset$, it is considered to be equivalent to $\{\mathtt{true}\}$, and when $\Delta = \emptyset$, it is equivalent to $\{\mathbf{false}\}$, and this follows directly from the rules.

The rules for propositional connectives:

$$\frac{\Gamma,\, \phi \Rightarrow \Delta}{\Gamma \Rightarrow \Delta,\, \neg\phi}\neg_R \qquad \frac{\Gamma \Rightarrow \Delta,\, \phi}{\Gamma,\, \neg\phi \Rightarrow \Delta}\neg_L$$

$$\frac{\Gamma \Rightarrow \Delta,\, \phi_1 \quad \Gamma \Rightarrow \Delta,\, \phi_2}{\Gamma \Rightarrow \Delta,\, \phi_1 \wedge \phi_2}\wedge_R \qquad \frac{\Gamma,\, \phi_1,\, \phi_2 \Rightarrow \Delta}{\Gamma,\, \phi_1 \wedge \phi_2 \Rightarrow \Delta}\wedge_L$$

$$\frac{\Gamma \Rightarrow \Delta,\, \phi_1,\, \phi_2}{\Gamma \Rightarrow \Delta,\, \phi_1 \vee \phi_2}\vee_R \qquad \frac{\Gamma,\, \phi_1 \Rightarrow \Delta \quad \Gamma,\, \phi_2 \Rightarrow \Delta}{\Gamma,\, \phi_1 \vee \phi_2 \Rightarrow \Delta}\vee_L$$

$$\frac{\Gamma,\, A \Rightarrow \Delta \quad \Gamma \Rightarrow \Delta,\, A}{\Gamma \Rightarrow \Delta}\mathrm{cut}.$$

Again, the soundness for all these rules is quite straightforward to argue. The cut rule is *admissible* in this proof system (that is, if a proof of a sequent $\mathcal{S}$ contains cut, then there exists a proof of $\mathcal{S}$ without cut). Even though it does not make the system more expressive (or "more complete"), it often makes the proof much shorter.

The rules for the quantifiers may require some explanations:

$$\frac{\Gamma \Rightarrow \Delta,\, \phi[a/x]}{\Gamma \Rightarrow \Delta,\, \forall x.\, \phi}\forall_R^a \qquad \frac{\Gamma,\, \phi[t/x] \Rightarrow \Delta}{\Gamma,\, \forall x.\, \phi \Rightarrow \Delta}\forall_L$$

$$\frac{\Gamma \Rightarrow \Delta,\, \phi[t/x]}{\Gamma \Rightarrow \Delta,\, \exists x.\, \phi}\exists_R \qquad \frac{\Gamma,\, \phi[a/x] \Rightarrow \Delta}{\Gamma,\, \exists x.\, \phi \Rightarrow \Delta}\exists_L^a$$

Here $\phi[t/x]$ means all free instances of x in ϕ are replaced by a term t, and a is a *new Skolem constant*, a constant that is not present anywhere in the conclusion or below it in the proof tree, and is originally uninterpreted by the model. We denote the condition that a must be a Skolem constant by adding it as a superscript to the rule's name.

The soundness of the $\forall_R^a$ rule relies on the fact that a is a new uninterpreted constant. Assume that the sequent in the premisses holds; that is, for any model M (which now has to interpret the new constant a),

$$M \models \bigwedge \Gamma \to \bigvee(\Delta,\, \phi[a/x]).$$

If $M \not\models \bigwedge \Gamma$, then $M_a \not\models \bigwedge \Gamma$ for any M_a which may differ from M only in the interpretation of a, but otherwise is the same, since a is new and does not occur in Γ. If $M \models \bigwedge \Gamma$, then we know that $M_a \models \bigvee(\Delta,\, \phi[a/x])$ for all M_a. But this is equivalent to saying that $M \models \bigvee(\Delta,\, \forall x.\, \phi)$, and since M was arbitrary, we know that the conclusion sequent also holds.

The intuition behind the soundness of the $\exists_R$ rule is much simpler. By definition of the existential quantifier, we want to find a value for x in every model that would make ϕ true. If we can construct a term t that would provide us with such a value in any model, then we can insert this term in place of x in ϕ and eliminate the quantifier, and the formula will still be true in every model. Note, that there are no restrictions on what term one can substitute for x in this rule (except for the types), and therefore, it is possible to use a term that would make the sequent in the premisses unprovable, even if the original sequent holds. This means that this rule is *not invertible*.

The other two quantifier rules are dual, and their soundness can be formally derived with the help of the following equalities:

$$\forall x.\, \phi \equiv \neg\exists\neg\phi, \qquad \exists x.\, \phi \equiv \neg\forall x.\, \neg\phi.$$

However, they can also be interpreted in a more intuitive way. The rule $\forall_L$ states that if $\forall x.\, \phi$ is assumed to hold in some model, then ϕ holds in the same model for any value of x, and in particular, the value provided by the term t. This rule is useful when one needs to use a concrete instance of a more general assumption. The conclusion sequent in the rule $\exists_L^a$ assumes that in any model we consider there exists a value for x that makes ϕ true. The rule gives us that value in the form of a Skolem constant a, which we can use later.

The proof system is sound, and is also known to be complete as it is. In addition, most of the rules in the Gentzen system are invertible. The only non-invertible rules are cut, $\exists_R$, and $\forall_L$. Because of these rules, if they are used in a proof of a theorem, and we arrive at a clearly invalid subgoal, we cannot say anything about the original theorem, as we may have made a bad choice of picking a formula in the cut rule, or an improper term in the quantifier instantiation rules.

Adding *interpreted functions and predicates*, such as the equality "$=$" and/or the arithmetic operators, will require additional rules that define the interpretation of these symbols, and it may make the proof system incomplete, as well as add more non-invertible rules.

2.2.3 User-Guided and Automated Proof Search

The simplest way to build a proof in an interactive prover (from the tool programmer's prospective) is to let the user guide the prover. Since we are building the proof from the main theorem towards axioms (in a bottom-up fashion), the user is supposed to tell the prover which rule to apply to that sequent. The prover verifies that the conclusion of that rule indeed matches the current sequent and generates the premisses as new subgoals. This process iterates until the proof tree is complete.

In this approach, the user does not have to know the proof beforehand, but rather, he or she constructs the proof as it develops, making decisions based on the subgoals generated by the prover. If a theorem is proven, the proof is guaranteed to be correct w.r.t. the proof system (which is assumed to be sound), since the user is not allowed to modify or introduce the sequents manually.

Although much better than a pencil and paper proof, sometimes this process can still be quite tedious. Moreover, parts of the proof, or even the entire proof may be long but conceptually simple, making the user repeat boring sequences of similar steps many times. In such cases, the proof search can be automated using what we call *strategies*, or meta-rules that specify how the inference rules must be applied. An example of a simple strategy can be the following: given a sequence of inference rules $r_1, r_2, \ldots, r_n$, try to apply r_1 to the current subgoal, and if it fails (the conclusion of r_1 does not match the current sequent), try r_2, etc., until we find r_i that does apply. Then we take the new subgoals and repeat the process for all of them. Another strategy may implement a simple *backtracking*: given a list of inference rules or other strategies, try to apply them one by one to the current subgoal, and if the rule or strategy does not prove the sequent (but not necessarily fail), then discard the partial proof subtree rooted at the current subgoal and try applying the next rule or strategy.

Notice, that the above examples of strategies do not require any knowledge of the internal structure of the rules or sequents. Thus, in our definition, a strategy is a way of automatic proof search that does not depend on the structure of rules and sequents, and can be used in an arbitrary proof system. The only dependency on the proof system can be in the names of the inference rules.

Although strategies are a very general and simple way to automate the proof search, for the same reason of generality they are rather inefficient, and only relatively simple proof search heuristics can be implemented with strategies.

A more flexible technique is what we call *tactics*. These should not be confused with tactics in Isabelle or HOL theorem provers, which are simply a way to implement an interactive backward proof search. Our notion of a tactic is similar to a strategy, only the decisions can be based on the internal structure of the sequent and the internal state of the tactic. Since tactics are highly proof-system dependent, they are not as universal as strategies, but can potentially be much more accurate and robust for specific problem domains.

Finally, there are many highly specialized techniques developed by the automated deduction community that involve *type inference* algorithms (such as those in the *Logical Framework* [Pfe94, Pfe99b, PS99]), multi-phase proof search with modified sequents, and other techniques that can achieve amazing results, mostly in the First-Order logic with or without equality, and sometimes even with arithmetic. Additionally, decision procedures (model checkers, arithmetic equation solvers, etc.) can be added as atomic proof rules and delegate the proof of decidable fragments to specialized efficient algorithms from the more general proof search engines.

In this work, we use only the first three approaches, the interactive user-guided proof search, strategies, and tactics. There are several reasons we do not go too far into the automated deduction. First, our framework is designed to be the basis for a research tool *SyMP* (see Chapter 4), and proof-system dependent techniques cannot be implemented in any meaningful way in the common core of the tool. Therefore, we have developed a few general strategies that can be used in any proof system.

Also, developing automated proof search in a new proof system often requires many experiments, and the best way to get an intuition behind it is to let the user construct the proofs interactively on several examples and then figure out some heuristics.

We also introduced the notion of tactics to be able to prototype some proof-system dependent proof search techniques, but in our experience, general strategies have so far been enough for those proofs that can be automated. The more advanced and specialized techniques can, in principal, be implemented as a part of the proof system itself (for instance, multi-phase proof search can be done by introducing several types of sequents and different groups of rules that work only on those sequents), together with tactics. However, this is already beyond the scope of this work, and we only concentrate on the ability to prototype new specialized proof systems and provide the interactive user interface with some degree of automation.

2.3 Existing Approaches Combining Model Checking and Theorem Proving

We have described the two mainstream approaches to formal verification, model checking and theorem proving. Both of these approaches have different strengths and weaknesses, and there has been much work in trying to combine them to be able to verify larger and more complex examples faster, more conveniently, and with less required user guidance. Below we discuss briefly a few most well-known techniques that either combine model checking and theorem proving in a certain way or borrow the ideas from both.

2.3.1 Abstraction and Symmetry

When the original model has symmetric components (e.g. an array of caches in a shared memory protocol), it is often possible to reduce the number of these components by reordering them dynamically [CJ95]. This way we can force the components that influence our specification to be always some particular selected components. Since the other components will not be important, we can then remove them from the model.

Alternatively, when the model is too large or even infinite, one may use *abstraction* to reduce the size of the model [Lon93]. Some constraints are removed from the original model and replaced by nondeterminism, making the model smaller (it has fewer constraints on the transitions), but with "more behaviors," so that the abstract model has more ways of reaching erroneous states. Therefore, if a safety property is true in the abstract model, then it must be true in the original model as well.

The two reduction techniques can be used together [McM98]. If the symmetry results from the type of a state variable being symmetric (that is, swapping any particular value of that type with any other value doesn't change the behavior of the model), then checking a property for all values over this type is equivalent to checking the property only for a small number of selected *representative* values. To remove the rest of the values of this type from the model, one can use abstraction which replaces the type by those few selected values and a $\perp$-value representing all the other values. In addition, the cone of influence reduction can often reduce the model significantly after abstraction and symmetry has been applied to the model.

Another breed of abstraction is *predicate abstraction* [SG97] where, given n predicates over the state variables, a possibly infinite model is replaced by the model with only n state bits, each bit representing the value of the respective predicate. Predicate abstraction is a very powerful tool, however, computing the transition relation of the abstracted model involves checking for validity many formulas involving the predicates. In the case of an infinite and sufficiently expressive model, the problem may become undecidable and very computationally intensive, if not requiring manual user guidance.

2.3.2 Assume-Guarantee Reasoning

Assume-Guarantee Reasoning [Pnu85, Lon93] is a variant of *compositional reasoning* when some properties are proven for each of the component, and then the property of interest is derived for their parallel composition. The properties of some components are proven under certain *assumptions* about other components, and then those components are proven to *guarantee* these assumptions. A typical rule for assume-guarantee reasoning is like this:

$$\frac{\mathbf{true} < M_1 > \psi \quad \psi < M_2 > \phi}{\mathbf{true} < M_1 \parallel M_2 > \phi \wedge \psi}.$$

Assume-guarantee traditionally has been used mostly in the model checking context, and therefore, is sometimes considered as part of model checking process. However, it is very close in spirit to *Modus Ponens* rule from theorem proving.

2.3.3 Inductive Proofs

The reduction techniques mentioned above come mostly from the model checking world. We now discuss some of the verification techniques from theorem proving.

From the point of view of temporal specifications, there are two types of induction that can be applied. One is *induction on time*, and the other is *induction on the data structures*.

Safety properties in theorem proving are often proven by induction on time. First, one proves that the property holds in the initial states (the base of the induction), and then, assuming that the property holds in some arbitrary state, one proves that all the states in its transition image also satisfy this property (inductive step). Since the original property is rarely inductive (not strong enough to satisfy the inductive step), it is often necessary to *strengthen* the invariant before it can be proven, and this is usually the hardest and the least automatic step in the verification. Nowadays there are a few tools that help compute inductive invariants automatically [BLO98, GS96].

While proving properties about complex or infinite data structures, one may need to use natural or structural induction within the current state of the system.

2.3.4 "Circular" Compositional Reasoning

A combination of assume-guarantee and induction on time yields the so-called *"circular" compositional reasoning* [MCon81, AL93, AH96, McM98]. In some systems with tightly coupled components, the dataflow goes back and forth among the components, and to prove the guarantees of one module, we need to assume some properties about the other, and *vice versa*. The classical assume-guarantee rule does not work in this case, since we cannot break the cyclic dependency. However, we can require that the assumptions ψ about the other components be considered only up to time t while proving the property ϕ at time $t+1$. This "strong" induction over time can be expressed as an LTL formula:

$$\neg(\psi \, \mathbf{U} \, \neg\phi),$$

which says that it is not the case that $\neg\phi$ holds at some point, and all the time until that ψ holds, which is propositionally equivalent to the inductive statement. The proof of this formula is then accomplished by model checking, either directly or after some abstraction.

2.3.5 Integration of Model Checkers into Theorem Provers

It is very natural and intuitively straightforward to use model checking techniques for proving a logical formula describing a finite-state property. In proof-theoretic terms, one can extend the notion of an axiom to any sequent that can be translated into a finite-state model checking problem and verified to be true. We will omit the concrete technical details of what sequents can be considered finite-state, but intuitively, any model checking problem can be translated into a higher-order logic formula. This formula will represent a finite-state problem, which potentially could be translated back into the original or equivalent model checking problem suitable for a model checking tool. It is conceivable that such a formula may also arise as a subgoal in a proof, in which case one can apply the same transformation and use a model checker to discharge the subgoal in one step.

This is precisely the approach taken, for example, by the *PVS* theorem prover [SOR93]. Another prover *ACL2* [KM00] uses BDDs to prove formulas over boolean variables. This is similar to the use of a model checker, and in fact, *PVS* has a similar set of rules for simplification of propositional formulas. A Stanford prover called *STeP* [BBC$^+$99] has a similar approach to incorporating a model checker into the main prover, however, since it is based on temporal logic and its input explicitly involves a model (which is later translated into a set of formulas), it has an easier time deciding the finiteness of a subgoal and extracting the model.

2.3.6 Integration of Theorem Proving Techniques into Model Checkers

Probably the best example of integrating theorem proving capabilities into what has previously been known as one of the "purest" model checkers is the Cadence version of *SMV* written by McMillan [McM98]. Originally, it was a relatively simple but very efficient (and the first of a kind to use BDDs) symbolic model checker [McM93]. Later, the author has added such capabilities as data abstraction along with symmetry reductions, circular compositional reasoning, induction on time and on data (natural induction, for instance), and much more. One of the most important differences between SMV and traditional theorem provers extended by a model checker is that the prover part of *SMV* has been developed on top of an existing model checker engine, and with the main purpose to reduce the original problem to a finite-state one as fast as possible, so that it can then be verified by this engine. Moreover, many reductions borrowed from theorem proving heavily use the model checking and related techniques, as it simplifies the implementation and increases efficiency.

While amazingly efficient and successful in practice, the tool has one serious drawback: it does not have any proof search mechanism (apart from automatic abstraction, which could be considered more a part of a extremely complex axiom rather than a proof step by itself), not even an interactive one. The user is expected to provide the complete proof beforehand. Fortunately, thanks to very powerful proof rules, the proofs are usually not as long and tedious as they could be in unspecialized general-purpose theorem provers.

Another tool that can be considered a model checker beefed up by theorem proving capabilities is the *Mocha* [AHM$^+$98] tool developed at Berkeley. It is an interactive environment for specification, simulation, and verification of concurrent systems. Most of the verification work is done by the built-in or external model checker, but to exploit the hierarchy of the design, *Mocha* uses assume-guarantee reasoning controlled by a prover module. Unlike *SMV*, this tool has an interactive construction of the proof, as well as some limited automated proof search capabilities. The scope of the prover is limited mostly to the assume-guarantee reasoning. However, it can potentially be extended to other aspects of verification process, and such extensibility is an explicit feature of the system.

Chapter 3

Unified Framework for Combining Model Checking and Theorem Proving

A very common class of verification problems often faced by the researchers in formal verification community is the class of *parameterized* designs, which on top of that include *datapaths* with large or even infinite sets of possible values. Such designs can also consist of *parallel components* composed together *synchronously* or *asynchronously*. The parameters of the design may include the number of parallel components, the width of the datapaths, scheduling algorithms, and many more.

In the presence of such a complicated design structure, formal verification requires non-trivial amount of user interaction and ingenuity, and the verification complexity may quickly become prohibitively large. More automatic techniques like finite-state model checking cannot be applied directly to these problems, since the search space is usually infinite or extremely large. Trying to find a proof in a "pure" theorem proving environment may quickly lose the user in the amount of detail, and the problem may quickly become unmanageable.

In this chapter, we present a new verification framework that enhances the notion of theorem proving with a more adequate problem representation and domain-specific reduction techniques. We also construct a concrete proof system within our framework specialized to the verification of hardware designs. This proof system serves as a theoretical basis for a theorem prover that effectively combines traditional notions of theorem proving and model checking, creating a balanced synergy of the two techniques and eliminating many of their performance bottlenecks.

Feature	Model Checking	Theorem Proving
Problem Representation	$M \models \phi$: "native" representation	$\models \Phi_M \to \phi$: HOL
Domain size	Finite	Infinite
Parameterized designs	Finite instances or abstraction (built externally)	Can be handled directly
Model transformations	Easy to do aggressive, domain-specific transformations	Limited, due to HOL representation
Finite State Enumeration	Extremely efficient upto $\approx 10^{20}$	Usually limited to $10^3 - 10^5$
Automation	Completely automatic	Often requires extensive user guidance

Figure 3.1: "Classical" features of model checking and theorem proving.

3.1 Model Checking vs. Theorem Proving

In order to construct a suitable verification formalism for our problem domain, we first examine the existing approaches, identify the desirable features and deficiencies of those, and then compose a new approach that would fit our purposes best.

Traditionally, there are two main branches in formal verification: model checking and theorem proving, and one would normally have to choose one or the other. The table on Figure 3.1 shows the most important "classical" features of model checking and theorem proving. It is clear from the table that model checking has an adequate, domain-specific representation of the problem: $M \models \phi$, where M is a model, explicitly separated from the specification ϕ. Both M and ϕ have a great freedom of internal representation, from explicit transition graph to BDDs or similar symbolic representation for the model, and from state automata to various temporal logics for the specification. Therefore, one has an option to pick the best suited combination for a specific problem domain. However, classical model checking techniques focus mainly on automatic verification. This limits most of the operations to model transformations, and the specification is usually not reduced. For example, an efficient and compact BDD representation, cone of influence reduction, various types of abstractions, and symmetry reduction are all targeted at reducing the size of the model. The reason for this is quite obvious, as the model is usually very large and complex, whereas the property could be very simple (e.g., *the system does not deadlock*). Due to the complexity of the model, it is almost impossible to expect a decent user guidance in the verification process, at least at the level of individual state transitions, which encourages the programmers to make the tools as automatic as possible. *Completeness* and *termination guarantee* of model checking has also been a pride of the field for a long time, as it enables the tool to guarantee the correctness w.r.t. a given property, or produce a counterexample otherwise. Unfortunately, the combination of termination and completeness implies decidability, which limits the scope of application of model checkers to finite models. Efficiency — another highly desirable feature — also limits the specification language to propositional temporal logics or small finite automata.

Theorem proving, on the other hand, has a nearly inverse situation. The higher order logic (HOL) as an input language (both for the model and specification) immediately rules out decidability, hence, completely automatic decision procedures are impossible from the start, and more attention is paid to expressiveness. Traditionally, most of the theorem provers are based on *Gentzen sequent*:

$$\Gamma \Longrightarrow \Delta,$$

where Γ and Δ are sets of HOL formulas. The sequent is interpreted as follows: whenever all of the formulas in Γ are true, at least one of the formulas in Δ must be true. One can think of the sequent as a special representation of a formula

$$\bigwedge \Gamma \to \bigvee \Delta,$$

which must be *valid*, or *generally true*. Modern theorem provers provide rich syntax for expressing (recursive) functions, complex datatypes, and even transition systems in terms of HOL, along with similarly beefed up proof systems capable of solving systems of non-linear arithmetic (in)equalities for conditionals in the program, automatic proof search, and much more. Since complete automation is impossible, it is often considered acceptable to expect the user to provide a lot of guidance to the system in return for a more general proof of a parameterized version of the design, or a design with infinite datatypes, or one where control is heavily data-dependent. Such general formulations of the problem are often far beyond the scope of any traditional model checkers. However, theorem provers have a hard time verifying designs with a relatively large and complex control graph. Since everything has to be translated into the HOL, the structure of the program may be lost, and any substantial transformation to the model, like the cone of influence reduction, may become very hard in this representation, unless special care is taken to keep the formulas in a very organized form. The formulas often become large and cumbersome, which quickly confuses even very sophisticated proof search engines, and the user is left with little choice but to guide the system manually through a number of carefully considered and tedious transformations until the problem is reduced to a size tractable by the available semi-automatic techniques.

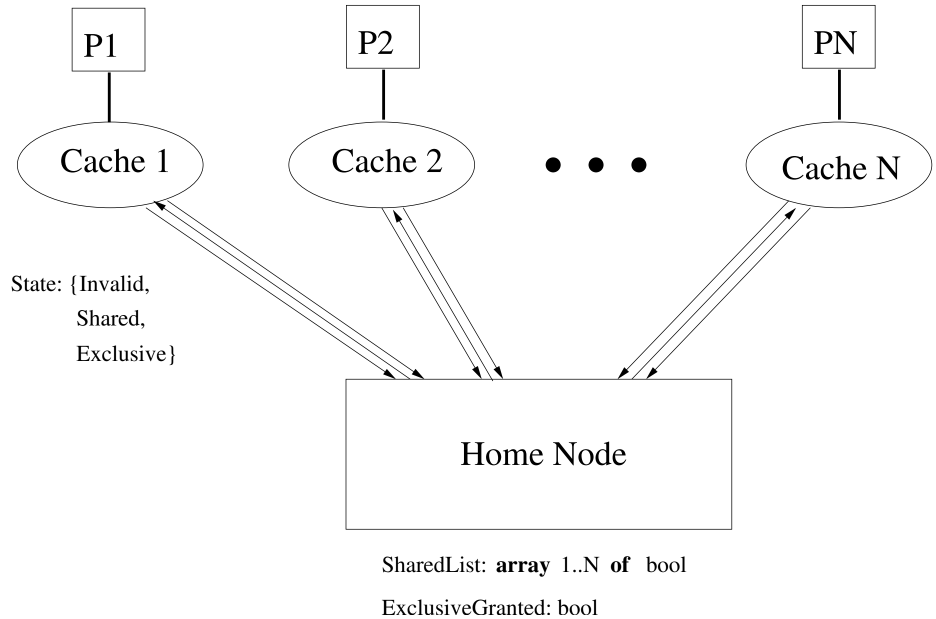

Figure 3.2: IBM Cache Coherence Protocol: Architecture

3.2 Motivating Example

To give a better intuition behind the ideas in this chapter, we provide a reference example of a *cache coherence protocol* originally developed at IBM for research purposes and given to us by Steven German. The design consists of N processors working in a shared memory model. Each processor has a private cache which communicates to the central memory module called *home node* through *communication channels with queues*. The processor + cache unit is called a *client*. The number of processors N, the number of cache lines in each cache L, the number of data bits stored in each line B, the memory address width A, and the length of the queues Q are the parameters to the design. To simplify the matters, we assume that $L = 1$, $B = A = 0$ (abstract away the data and address completely), $Q = 1$, and the only parameter left undefined is N. Since each memory address is independent of the others, abstracting away B, A, and L will not lose the generality. The queue length does reduce the generality, but it is not important for demonstrating our ideas in this work, and we still have one parameter N left to deal with. The overall architecture of the design is shown in Figure 3.2. The home node keeps track of which caches have the data and whether an exclusive copy of the data has been granted to any of the caches, and it is responsible for ensuring the coherence of the caches. Different types of messages are routed through different one-directional channels between the home node and the clients. The concrete details of the protocol are given in Section 5.1. Here we concentrate only on the high-level view of the verification process and will not need to know the implementation of the actual protocol.

As for any cache coherence protocol, we want to verify that each processor sees the memory state consistently with all the other processors; we call this property *coherence*. A part of this property is to ensure that only one processor is allowed to write to the same memory address at a time, which is a *mutual exclusion property*. We formulate it as follows: if some cache is in the Exclusive state, then all the other caches must be in the Invalid state.

This example has several features that makes it a good representative in the class of problems we are going to address. First of all, it is parameterized by the number of parallel components, and we have to verify it for all the values of the parameter, so the model can be potentially infinite.

On top of that, this design has an *unbounded nondeterminism*: when several caches send requests at the same time, the home node must pick one of them to service next, and we implement the scheduling algorithm as a nondeterministic choice. Since the number of caches is not bounded (it is parameterized), the nondeterministic choice can also get arbitrarily large. In addition, the queues in the communication channels make the design much harder to verify using traditional model checking techniques due to the BDD size explosion typical for queue-like data structures.[1] Finally, to express the property of mutual exclusion in the same parameterized manner as the design itself, we have to use at least the first-order temporal logic. For instance, in the first-order CTL the specification can be written as follows:

$$\mathbf{AG}\,(\forall c_1,\, c_2.\, c_1 \neq c_2 \wedge \mathrm{cache}(c_1) = \mathrm{Exclusive} \rightarrow \mathrm{cache}(c_2) = \mathrm{Invalid}).$$

All these issues can be summarized as three main problems that we have to solve:

1. Provide an appropriate formalism for the representation of parameterized potentially infinite designs with unbounded nondeterminism;

2. Find expressive enough specification language (or the *logic*) for writing the properties; and

3. Develop verification methodologies for such designs and specifications.

The easiest part is to find the logic: at least for the given example the first-order CTL is sufficient. In fact, it turns out to be sufficient (but not necessarily the most convenient)

[1] In this example, the queue depth is only 1, so the queues do not cause BDDs to blow up, but the generalization of this example to longer, or even arbitrary queues would exhibit such behavior, and could be another interesting example to verify.

in most of other examples we have tried. Later in this chapter we introduce even more powerful logics like first-order branching time and linear time μ-calculus.

One of the important issues in representing parameterized designs is the adequacy of the representation. That is, we would like to preserve as much information about the model as we can, so that the verification algorithms can exploit it later. Therefore, a carefully designed input language adequate for the problem domain is necessary. Our solution to this problem is the input language of the default proof system in *SyMP* (see Section 4.2).

The rest of this chapter is devoted mostly to building a formalism that utilizes both model checking and theorem proving techniques and makes it possible to verify the examples like the one above.

3.3 Combining Model Checking and Theorem Proving

3.3.1 New Sequent

Ideally, one would like to pick the best features out of both approaches, somehow combine them together, and most of the drawbacks should cancel out automatically in the resulting framework. For example, lifting the decidability restriction in model checking and adding the expressiveness of HOL should solve the problem with expressiveness. At the same time, clear separation of the model and the specification should, in principle, lead to more efficient reductions and proof search procedures. The only question is how actually to do such a combination in a way that neither approach loses too many of its strong points.

Since theorem proving is a more general approach, and we want to maintain its full generality, let us look at it closer and see what causes it to break on complex examples. One reason, in our opinion, is the inadequate representation of the problem domain. Since in formal verification we are interested in verifying a complex model, let us add this model explicitly to the Gentzen sequent with the following semantics:

$$M;\, \Gamma \Longrightarrow \Delta \text{ holds iff } M \models \bigwedge \Gamma \rightarrow \bigvee \Delta. \tag{3.1}$$

The model in this new sequent can be represented in the same way as in a model checker, in its most suitable representation. In addition, we allow the model M to be infinite, since we are no longer limited by the finite-state requirement of model checking.

Next step is to define a proof system for this new sequent, since the original Gentzen system is no longer directly applicable. As the first step, however, we can rewrite all the inference rules in the standard Gentzen proof system to carry the model along without modifying it. Since a proven formula in the Gentzen system is valid in all models, it will also be valid in the given model M, the soundness is trivially preserved. At the same time, we did not sacrifice any generality of theorem proving, as we preserved the entire proof system for HOL. Next, notice that if the model is finite and the formula is propositional, we are ready to apply model checking techniques directly to the sequent and, thus, may prove the sequent in one step. Similarly, we can easily apply virtually any transformation to the model that exists in the model checking world; for example, a cone of influence reduction, an abstraction, etc. These transformations will, in principle, be no harder to do than in any other model checker, since we have the model and the specification explicitly separated and represented adequately for the problem domain.

However, the proof system we have built so far suffers from two major drawbacks. First, the specification language, although it is the full HOL, can only refer to the *initial state* of the model M, while the most interesting properties are *temporal*. And second, we lose the ability to employ the full power of theorem proving on the model itself, leaving it completely to the model checking transformations. Hence, our sequent and, consequently, the proof system need to be revised.

Let us define the new version of our sequent as follows:

$$M;\, \Sigma;\, \Gamma \vdash \Delta,$$

where $M = (S_M,\, \to_M\;\; I_M)$ is a model with the set of states S_M, transition relation $\to_M \subseteq S_M \times S_M$, and the set of initial states $I_M \subseteq S_M$, and Σ, Γ, and Δ are sets of first-order μ-calculus formulas. Σ represents the set of *invariant constraints*, that is, the properties that must hold in every reachable state of M; Γ is the set of *assumptions* at the initial state of M; and Δ is the set of *conclusions*. A sequent *holds* iff for any initial state of M at least one of the formulas in Δ holds under the invariant constraints Σ and the assumptions in Γ. More formally, the sequent

$$M;\, \Sigma;\, \Gamma \vdash \Delta$$

holds iff

$$M|_\Sigma \models \bigwedge \Gamma \to \bigvee \Delta,$$

and $M|_\Sigma = (S',\, \to',\, I')$ is obtained from M by restricting the set of states to only those that satisfy all formulas in Σ. That is,

- $S' = \{s \mid M,\, s \models \bigwedge \Sigma\}$;
- $\to' = \to \cap S' \times S'$;

- $I' = I \cap S'$.

This sequent allows us to express temporal properties of the model and, as we will see later, allows theorem proving techniques to be applied to some parts of the model, effectively tightly combining model checking and theorem proving techniques in one single and uniform framework.

The new sequent differs from our original sequent (3.1) in two ways: the underlying logic is temporal (first-order μ-calculus vs. HOL) and there is an extra set of invariant assumptions Σ. The latter is needed for technical reasons in the rules for eliminating the fixpoint operators of the logic.

3.4 The Logic

Our logic is the usual modal μ-calculus [Koz83] with the addition of universal and existential quantifiers over object variables.

3.4.1 Syntax

Formally, define terms, predicates, and formulas as follows.

$$
\begin{array}{rcl}
\text{Term} & ::= & c \,|\, x \,|\, v \,|\, f(t_1, \dots, t_n) \\
\text{Predicate} & ::= & X \,|\, P(t_1, \dots, t_n) \\
\text{Formula} & ::= & \text{Predicate} \,|\, \phi_1 \wedge \phi_2 \,|\, \phi_1 \vee \phi_2 \,|\, \neg\phi \,|\, \phi_1 \to \phi_2 \\
& & |\, \diamond \phi \,|\, \Box\phi \\
& & |\, \forall x.\, \phi \,|\, \exists x.\, \phi \\
& & |\, \mu X.\, \phi^+(X) \,|\, \nu X.\, \phi^+(X)
\end{array}
$$

Here $\phi^+(X)$ means that ϕ is *positive* on X, that is, X occurs in ϕ only in the scope of even number of negations (where left hand side of implication is considered a negation in this definition). The terms are composed of constants c, logical (or object) variables x, state variables v, and function applications $f(t_1, \dots, t_n)$. In our framework all terms are typed, and the types of terms in the application have to be consistent with the function's type. In other words, we assume all our terms to be well-typed, without giving a formal definition of it for now.

3.4.2 Semantics

The semantics is exactly the same as for the propositional μ-calculus, and the quantifiers are interpreted as infinite conjunction and disjunction. For a Kripke structure $M = (S, \rightarrow, I)$,[2] a formula ϕ is interpreted as a subset of S where it is true. We denote this subset by $[\![\phi]\!]_M e$, where e is an *environment* that assigns interpretation to propositional and object variables. We may omit the subscript M when unambiguous.

$$
\begin{array}{rcl}
[\![X]\!]e & = & e(X) \\
[\![\phi_1 \wedge \phi_2]\!]e & = & [\![\phi_1]\!]e \cap [\![\phi_2]\!]e \\
[\![\phi_1 \vee \phi_2]\!]e & = & [\![\phi_1]\!]e \cup [\![\phi_2]\!]e \\
[\![\neg\phi]\!]e & = & \overline{[\![\phi]\!]e} \\
[\![\phi_1 \rightarrow \phi_2]\!]e & = & \overline{[\![\phi_1]\!]e} \cup [\![\phi_2]\!]e \\
[\![\Box\phi]\!]e & = & \{s \,|\, \forall s'.\, s \rightarrow s' \text{ implies } s' \in [\![\phi]\!]e\} \\
[\![\Diamond\phi]\!]e & = & \{s \,|\, \exists s' \in S.\, s \rightarrow s' \text{ and } s' \in [\![\phi]\!]e\} \\
[\![\forall x.\, \phi]\!]e & = & \bigcap_{a \in D_x} [\![\phi]\!]e[x \leftarrow a] \\
[\![\exists x.\, \phi]\!]e & = & \bigcup_{a \in D_x} [\![\phi]\!]e[x \leftarrow a] \\
[\![\mu X.\, \phi(X)]\!]e & = & \bigcap\{S \,|\, [\![\phi(X)]\!]e[X \leftarrow S] \subseteq S\} \\
[\![\nu X.\, \phi(X)]\!]e & = & \bigcup\{S \,|\, S \subseteq [\![\phi(X)]\!]e[X \leftarrow S]\}
\end{array}
$$

A predicate $P(t_1, \ldots, t_n)$ is interpreted as a set of states where it is true:

$$[\![P(t_1, \ldots, t_n)]\!]e = \{s \,|\, [\![P]\!]^s e([\![t_1]\!]^s e, \ldots, [\![t_n]\!]^s e) \text{ is true}\},$$

where $[\![t]\!]^s e$ is a value of the term t in a state s. Thus, we treat an n-ary predicate symbol as an n-ary function from $D_1 \times \cdots \times D_n$ to $\{\mathtt{true}, \mathbf{false}\}$. The constants have a fixed interpretation $[\![c]\!]^s_M e$ in the model M, and the other terms are interpreted as follows:

$$
\begin{array}{rcl}
[\![x]\!]^s e & = & e(x) \\
[\![v]\!]^s e & = & s(x) \\
[\![f(t_1, \ldots, t_n)]\!]^s e & = & ([\![f]\!]^s e)([\![t_1]\!]^s e, \ldots, [\![t_n]\!]^s e),
\end{array}
$$

where the interpretation of the function symbol f is $[\![f]\!]^s e$, which is a function from $D_1 \times \cdots \times D_n$ to D in M.

[2] The labeling function L_M is omitted here because it is implicitly defined by the interpretation of the predicates and terms in M.

3.5 Proof System for the First-Order Branching Time μ-calculus

To simplify the notation for our proof system, we will write $M \models \phi$ for $I_M \subseteq [\![\phi]\!]_M \cdot$, where I_M is the set of initial states of M and "$\cdot$" is the empty environment; that is, the formula ϕ must be closed (without free variables). Note, however, that ϕ may have some *uninterpreted constants* (e.g. Skolem constants); then $M \models \phi$ means $M' \models \phi$ for all M' that may differ from M in the interpretation of such constants, but otherwise is the same as M.

Our proof system follows Gentzen style sequent calculus with a sequent, as defined in Section 3.3, of the following form:

$$M;\ \Sigma;\ \Gamma \vdash \Delta,$$

where Σ, Γ and Δ are lists of formulas. Recall that the meaning of the sequent, intuitively, is that one of the formulas in Δ is true in all of the initial states that satisfy all formulas in Γ, assuming that all of the formulas in Σ are global invariants of the model (that is, all formulas in Σ are true in the entire set of reachable states).

Formally, we want the following theorem to be true:

Theorem 3.5.1. ***(Soundness)*** *Given a model M, if $M;\ \Sigma;\ \Gamma \vdash \Delta$ is derivable in the proof system, then*

$$M|_\Sigma \models \bigwedge \Gamma \rightarrow \bigvee \Delta.$$

The formal proof of this theorem is in appendix. We do not provide it here mostly because the soundness of most of the rules is quite straightforward. The less obvious rules are supplied with an informal description explaining why they should be sound.

3.5.1 Inference Rules

In some rules we introduce new parameters (or *Skolem constants*), and such rules are marked by the constants in the superscript. For instance, $\forall_R^a$ is universal quantifier elimination on the right that introduces a new parameter a. The parameter is *new* means it does not occur anywhere in the proof below the line.

In the basic proof system described in this section the invariant assumptions Σ are only added by the fixpoint rules and the Cut_I rule. Later, when we consider concrete examples, we will be expanding the set of rules to accommodate efficient reductions and shortcuts in the verification process, and some of those new rules will also modify Σ. All of the other rules simply carry Σ through without modifying it, but may potentially use formulas from Σ.

Axioms. The only axioms in our system are the *initial sequent*s:

$$\frac{}{M;\, \Sigma;\, \Gamma,\, \phi \vdash \Delta,\, \phi}\mathrm{Init} \qquad \frac{}{M;\, \Sigma,\, \phi;\, \Gamma \vdash \Delta,\, \phi}\mathrm{Init}_I.$$

The Init_I rule is derivable from Init and weakening, but we also introduce it explicitly.

The rules for model checking. The direct model checking rule collects some formulas from Σ, Γ, and Δ in one large implication which is then passed to a model checker:

$$\frac{}{M;\, \Sigma;\, \Gamma \vdash \Delta}\mathrm{MC} \quad \text{provided that} \quad M|_\Sigma \models \bigwedge \Gamma \to \bigvee \Delta.$$

Here $M|_\Sigma$ is the model M restricted to the set of states that satisfy all the formulas in Σ (defined formally in appendix A), and the side condition is a finite-state model checking problem. In practice, the side condition is checked with any available finite-state model checking techniques.

Propositional rules. Propositional rules are very similar to the rules for the classical logic:

$$\frac{M;\, \Sigma;\, \Gamma,\, \phi \vdash \Delta}{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \neg\phi}\neg_R \qquad \frac{M;\, \Sigma;\, \Gamma \vdash \phi}{M;\, \Sigma;\, \Gamma,\, \neg\phi \vdash \Delta}\neg_L$$

$$\frac{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \phi_1 \quad M;\, \Sigma;\, \Gamma \vdash \Delta,\, \phi_2}{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \phi_1 \wedge \phi_2}\wedge_R \qquad \frac{M;\, \Sigma;\, \Gamma,\, \phi_1,\, \phi_2 \vdash \Delta}{M;\, \Sigma;\, \Gamma,\, \phi_1 \wedge \phi_2 \vdash \Delta}\wedge_L$$

$$\frac{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \phi_1,\, \phi_2}{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \phi_1 \vee \phi_2}\vee_R \qquad \frac{M;\, \Sigma;\, \Gamma,\, \phi_1 \vdash \Delta \quad M;\, \Sigma;\, \Gamma,\, \phi_2 \vdash \Delta}{M;\, \Sigma;\, \Gamma,\, \phi_1 \vee \phi_2 \vdash \Delta}\vee_L$$

$$\frac{M;\, \Sigma;\, \Gamma,\, \phi_1 \vdash \Delta,\, \phi_2}{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \phi_1 \to \phi_2}\to_R \qquad \frac{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \phi_2 \quad M;\, \Sigma;\, \Gamma,\, \phi_1 \vdash \Delta}{M;\, \Sigma;\, \Gamma,\, \phi_1 \to \phi_2 \vdash \Delta}\to_L$$

There is only one propositional rule for the invariant:

$$\frac{M;\, \Sigma,\, \phi_1,\, \phi_2;\, \Gamma \vdash \Delta}{M;\, \Sigma,\, \phi_1 \wedge \phi_2;\, \Gamma \vdash \Delta}\wedge_I$$

Weakening and Strengthening rules. It is obviously true that if we proved a property about a formula for a larger set of initial states, then it holds for a smaller set of states; and the opposite is true about the assumptions. This essentially means that adding a formula either to the left or to the right will preserve the validity of the sequent:

$$\frac{M;\, \Sigma;\, \Gamma \vdash \Delta}{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \phi} w_R \qquad \frac{M;\, \Sigma;\, \Gamma \vdash \Delta}{M;\, \Sigma;\, \Gamma,\, \phi \vdash \Delta} s_L.$$

Notice, that there is no a strengthening rule for the invariants Σ, since strengthening the invariant may invalidate reachability formulas like $\mathbf{EF}\,\psi$.

If we can prove some property in the entire model, then it is obviously true in the initial states:

$$\frac{M';\, \Sigma;\, \Gamma \vdash \Delta}{M;\, \Sigma;\, \Gamma \vdash \Delta} s_{\text{Init}},$$

where $M' = (S_M,\, \rightarrow_M,\, I')$, that is, M' is the same as M except that the set of initial states is replaced with I', which is the set of all states of M satisfying all the invariant assumptions in Σ.

We also allow to move an invariant of the model into the initial state predicates. This rule is actually invertible, but we still consider it a variant of a weakening rule:

$$\frac{M;\, \Sigma,\, \phi;\, \Gamma,\, \phi \vdash \Delta}{M;\, \Sigma,\, \phi;\, \Gamma \vdash \Delta} w_I$$

In order to be able to use an assumption more than once, we allow copying of the initial state formulas and invariants:

$$\frac{M;\, \Sigma;\, \Gamma,\, \phi,\, \phi \vdash \Delta}{M;\, \Sigma;\, \Gamma,\, \phi \vdash \Delta} \text{Copy}_L \qquad \frac{M;\, \cdot;\, \Gamma \vdash \Delta,\, \phi,\, \phi}{M;\, \cdot;\, \Gamma \vdash \Delta,\, \phi} \text{Copy}_R$$

$$\frac{M;\, \Sigma,\, \phi,\, \phi;\, \Gamma \vdash \Delta}{M;\, \Sigma,\, \phi;\, \Gamma \vdash \Delta} \text{Copy}_I$$

Monotonicity rules. The monotonicity properties are stated for an arbitrary formula $\psi^+(X)$ which is positive on X (and we write a "+" in the superscript of ψ to indicate that):

$$\frac{M' = (S,\, \rightarrow,\, S);\, \Sigma;\, \phi_1 \vdash \phi_2}{M = (S,\, \rightarrow,\, I);\, \Sigma;\, \Gamma,\, \psi^+(\phi_1) \vdash \Delta,\, \psi^+(\phi_2)} \text{Mono}$$

Where M' is the same as M except that the set of initial states is the entire S.

Case Splitting Rules. We have two types of case split in our system: finite (the usual *cut* rule) and its infinite version. In the cut rule we introduce a new formula and split the cases on whether it is true or false:

$$\frac{M;\, \Sigma;\, \Gamma,\, \phi \vdash \Delta \quad M;\, \Sigma;\, \Gamma \vdash \Delta,\, \phi}{M;\, \Sigma;\, \Gamma \vdash \Delta}\mathrm{Cut}$$

$$\frac{M;\, \Sigma,\, \phi;\, \Gamma \vdash \Delta \quad M;\, \Sigma;\, \Gamma \vdash \Delta,\, \mathbf{AG}\,\phi}{M;\, \Sigma;\, \Gamma \vdash \Delta}\mathrm{Cut}_I$$

In the rule Cut_I, the formula $\mathbf{AG}\,\phi$ is an abbreviation for $\nu X.\phi \wedge \Box X$, and it means that ϕ holds in every state reachable from the set of initial states.

It is common in verification that we need to consider the cases over the values of a certain term. Although we can already do it with the existing rules (using finite Cut, for example), we introduce a new (redundant) rule to simplify the proofs:

$$\frac{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \forall x.\, t = x \to \phi}{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \phi}\forall\mathrm{Case}_R,$$

where x is not a free variable of t or ϕ.

Rules for Modalities. If we have proven something at the current state, we may step back in time and claim that what we have proven will be true at the next step (provided the initial states of M do not interfere with the assumptions):

$$\frac{M';\, \Sigma;\, \Gamma \vdash \phi}{M;\, \Sigma;\, \Box\Gamma \vdash \Box\phi}\Box_{LR} \qquad \frac{M';\, \Sigma;\, \phi \vdash \Delta}{M;\, \Sigma;\, \Diamond\phi \vdash \Diamond\Delta}\Diamond_{LR},$$

where $M' = (S_M,\, \to,\, S_M)$ (eliminating the interference of the initial states), and $\Box\Gamma \equiv \Box\phi_1, \ldots, \Box\phi_n$ and $\Diamond\Delta \equiv \Diamond\psi_1, \ldots, \Diamond\psi_n$ when $\Gamma = \phi_1, \ldots, \phi_n$ and $\Delta = \psi_1, \ldots, \psi_n$. These rules are derivable from the monotonicity rules and some rules for modalities, but it is very useful to have them explicitly.

We need a few rules that define some properties of $\Box$ and $\Diamond$ with respect to propositional connectives:

$$\frac{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \Box\phi_1 \wedge \Box\phi_2}{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \Box(\phi_1 \wedge \phi_2)}\Box\wedge_R \qquad \frac{M;\, \Sigma;\, \Gamma,\, \Box\phi_1 \wedge \Box\phi_2 \vdash \Delta}{M;\, \Sigma;\, \Gamma,\, \Box(\phi_1 \wedge \phi_2) \vdash \Delta}\Box\wedge_L$$

$$\frac{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \Diamond\phi_1 \vee \Diamond\phi_2}{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \Diamond(\phi_1 \vee \phi_2)}\Diamond\vee_R \qquad \frac{M;\, \Sigma;\, \Gamma,\, \Diamond\phi_1 \vee \Diamond\phi_2 \vdash \Delta}{M;\, \Sigma;\, \Gamma,\, \Diamond(\phi_1 \vee \phi_2) \vdash \Delta}\Diamond\vee_L$$

Dualities between $\Box$ and $\Diamond$, or *negation propagation* rules:

$$\frac{M;\, \Sigma;\, \Gamma,\, \Diamond\phi \vdash \Delta}{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \Box\neg\phi}\neg\Box_R \qquad \frac{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \Diamond\phi}{M;\, \Sigma;\, \Gamma,\, \Box\neg\phi \vdash \Delta}\neg\Box_L$$

$$\frac{M;\,\Sigma;\,\Gamma,\,\Box\phi\,\vdash\,\Delta}{M;\,\Sigma;\,\Gamma\,\vdash\,\Delta,\,\diamond\neg\phi}\neg\diamond_R \qquad \frac{M;\,\Sigma;\,\Gamma\,\vdash\,\Delta,\,\Box\phi}{M;\,\Sigma;\,\Gamma,\,\diamond\neg\phi\,\vdash\,\Delta}\neg\diamond_L$$

And the last few rules that are needed for the completeness w.r.t. modalities:

$$\frac{M;\,\Sigma;\,\Gamma\,\vdash\,\Delta,\,\diamond\phi_1\wedge\Box\phi_2}{M;\,\Sigma;\,\Gamma\,\vdash\,\Delta,\,\diamond(\phi_1\wedge\phi_2)}\diamond\wedge_{R_1} \qquad \frac{M;\,\Sigma;\,\Gamma\,\vdash\,\Delta,\,\Box\phi_1\wedge\diamond\phi_2}{M;\,\Sigma;\,\Gamma\,\vdash\,\Delta,\,\diamond(\phi_1\wedge\phi_2)}\diamond\wedge_{R_2}$$

$$\frac{M;\,\Sigma;\,\Gamma,\,\diamond(\phi_1\wedge\phi_2)\,\vdash\,\Delta}{M;\,\Sigma;\,\Gamma,\,\diamond\phi_1\wedge\Box\phi_2\,\vdash\,\Delta}\diamond\wedge_{L_1} \qquad \frac{M;\,\Sigma;\,\Gamma,\,\diamond(\phi_1\wedge\phi_2)\,\vdash\,\Delta}{M;\,\Sigma;\,\Gamma,\,\Box\phi_1\wedge\diamond\phi_2\,\vdash\,\Delta}\diamond\wedge_{L_2}$$

The last group of rules relies on the following valid formula:

$$\diamond\phi_1\wedge\Box\phi_2\rightarrow\diamond(\phi_1\wedge\phi_2).$$

Intuitively, for the formula $\phi_1\wedge\phi_2$ to hold in *some* next state it is sufficient to show that ϕ_1 holds in *some* next state ($\diamond\phi_1$) and ϕ_2 holds in *all* next states ($\Box\phi_2$). The formal proof of this fact can be easily reconstructed from the definitions of the modalities.

Rules for Quantifiers. Quantifiers are handled exactly the same way as in the classical Gentzen-style proof system:

$$\frac{M;\,\Sigma;\,\Gamma\,\vdash\,\Delta,\,\phi(a)}{M;\,\Sigma;\,\Gamma\,\vdash\,\Delta,\,\forall x.\,\phi(x)}\forall_R^a \qquad \frac{M;\,\Sigma;\,\Gamma,\,\phi(t)\,\vdash\,\Delta}{M;\,\Sigma;\,\Gamma,\,\forall x.\,\phi(x)\,\vdash\,\Delta}\forall_L$$

$$\frac{M;\,\Sigma;\,\Gamma\,\vdash\,\Delta,\,\phi(t)}{M;\,\Sigma;\,\Gamma\,\vdash\,\Delta,\,\exists x.\,\phi(x)}\exists_R \qquad \frac{M;\,\Sigma;\,\Gamma,\,\phi(a)\,\vdash\,\Delta}{M;\,\Sigma;\,\Gamma,\,\exists x.\,\phi(x)\,\vdash\,\Delta}\exists_L^a$$

Rules for induction. In this rule we treat the domain of natural numbers in its standard interpretation (all the other rules are over uninterpreted domains), and introduce the induction with the base 0 and step size 1:

$$\frac{M;\,\cdot;\,\Gamma\,\vdash\,\Delta,\,\phi(0) \quad M;\,\cdot;\,\Gamma,\,\phi(a)\,\vdash\,\Delta,\,\phi(a+1)}{M;\,\Sigma;\,\Gamma\,\vdash\,\Delta,\,\forall x\in\mathcal{N}.\,\phi(x)}\mathrm{Ind}_R^a.$$

This rule can be generalized to any domain D with a well-founded partial order $<$:

$$\frac{M;\,\Sigma;\,\Gamma,\,\forall y\in D.\,y<a\rightarrow\phi(y)\,\vdash\,\Delta,\,\phi(a)}{M;\,\Sigma;\,\Gamma\,\vdash\,\Delta,\,\forall x\in D.\,\phi(x)}\mathrm{wf\_Ind}_R^a.$$

Notice that in the latter rule there is no base case. This is sound because the partial order $<$ is well-founded (doesn't have infinite descending chains), and therefore, the base case is when a doesn't have a predecessor, in which case $\forall y\in D.\,y<a\rightarrow\phi(y)$ will be vacuously true, and we have to prove $\phi(a)$ without an inductive hypothesis.

Rules for Fixpoints. The fixpoint rules are divided into two groups. The first group consists of rules that "unroll" the fixpoints. These basically state that the fixpoint formulas are fixpoints of themselves:

$$\frac{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \phi^+(\nu X.\, \phi^+(X))}{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \nu X.\, \phi^+(X)}\nu\mathrm{fix}_R$$

$$\frac{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \phi^+(\mu X.\, \phi^+(X))}{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \mu X.\, \phi^+(X)}\mu\mathrm{fix}_R$$

$$\frac{M;\, \Sigma;\, \Gamma,\, \phi^+(\nu X.\, \phi^+(X)) \vdash \Delta}{M;\, \Sigma;\, \Gamma,\, \nu X.\, \phi^+(X) \vdash \Delta}\nu\mathrm{fix}_L$$

$$\frac{M;\, \Sigma;\, \Gamma,\, \phi^+(\mu X.\, \phi^+(X)) \vdash \Delta}{M;\, \Sigma;\, \Gamma,\, \mu X.\, \phi^+(X) \vdash \Delta}\mu\mathrm{fix}_L$$

$$\frac{M;\, \Sigma,\, \phi^+(\nu X.\, \phi^+(X));\, \Gamma \vdash \Delta}{M;\, \Sigma,\, \nu X.\, \phi^+(X);\, \Gamma \vdash \Delta}\nu\mathrm{fix}_I$$

$$\frac{M;\, \Sigma,\, \phi^+(\mu X.\, \phi^+(X));\, \Gamma \vdash \Delta}{M;\, \Sigma,\, \mu X.\, \phi^+(X);\, \Gamma \vdash \Delta}\mu\mathrm{fix}_I$$

The other group of rules eliminates fixpoint operators, and these are the only rules in the basis of our proof system, besides the Cut_I rule, that introduce new formulas into the set of invariant assumptions Σ. These rules follow Tarski's semantics of the fixpoints, where the greatest fixpoint is the union of all pre-fixpoints, and the least fixpoint is the intersection of all the post-fixpoints:

$$[\![\mu X.\, \phi(X)]\!]e = \bigcap\{Q \mid [\![\phi(X)]\!]e[X \leftarrow Q] \subseteq Q\}$$

$$[\![\nu X.\, \phi(X)]\!]e = \bigcup\{Q \mid Q \subseteq [\![\phi(X)]\!]e[X \leftarrow Q]\}.$$

Showing that a sequent $M;\, \Sigma;\, \Gamma \vdash \Delta,\, \mu X.\, \phi(X)$ or $M;\, \Sigma;\, \Gamma \vdash \Delta,\, \nu X.\, \phi(X)$ holds is equivalent to showing that the set of initial states I of M restricted by Γ, Δ, and Σ is a subset of $[\![\mu X.\, \phi(X)]\!]e$ or $[\![\nu X.\, \phi(X)]\!]e$ respectively:

$$I \subseteq [\![\mu X.\, \phi(X)]\!]e$$

and

$$I \subseteq [\![\nu X.\, \phi(X)]\!]e.$$

Expanding the definition of the fixpoints, we have:

$$\forall Q.\, [\![\phi(X)]\!]e[X \leftarrow Q] \subseteq Q \Longrightarrow I \subseteq Q,$$

for the μ-fixpoint, and

$$\exists Q.\, Q \subseteq [\![\phi(X)]\!]e[X \leftarrow Q] \wedge I \subseteq Q$$

for the ν-fixpoint. Thus, proving that the greatest fixpoint includes all the states from I requires finding a *witness* Q for the existential quantifier. If we can find a formula ψ such that $Q = [\![\psi]\!]e$ is such a witness, then checking that $I \subseteq Q$ is the same as proving a sequent

$$M;\, \Sigma;\, \Gamma \vdash \Delta,\, \psi.$$

Similarly, $Q \subseteq [\![\phi(X)]\!]e[X \leftarrow Q]$ is equivalent to proving that ψ implies $\phi(\psi)$ in *every* state of the model M, or formally:

$$M';\, \Sigma;\, \psi \vdash \phi(\psi),$$

where $M' = (S,\, \rightarrow,\, S)$ is the same as $M = (S,\, \rightarrow,\, I)$, only the set of initial states of M' is the entire set of states S.

Putting it all together, we obtain a rule for eliminating ν-fixpoint on the right:

$$\frac{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \psi \quad M' = (S,\, \rightarrow,\, S);\, \Sigma;\, \psi \vdash \phi^{+}(\psi)}{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \nu X.\, \phi^{+}(X)}\nu_R$$

The '$+$' superscript indicates that $\phi^{+}(X)$ must be positive in X.

The rule for μ-fixpoint on the right is derived in a similar fashion from its semantical characterization:

$$\forall Q.\, [\![\phi(X)]\!]e[X \leftarrow Q] \subseteq Q \Longrightarrow I \subseteq Q.$$

This time we have to consider *all possible* sets of states Q to check this formula, and we capture this by introducing a *fresh uninterpreted predicate* α (it is appropriate to think of it as a *Skolem constant*, only for predicates). Now this formula is equivalent to proving that $I \subseteq [\![\alpha]\!]e$ for every interpretation of α such that $\phi(\alpha) \rightarrow \alpha$ holds in every state of the model M. This is where we make use of the invariant assumptions Σ: the above statement can now be stated as the following sequent:

$$M;\, \Sigma,\, \phi(\alpha) \rightarrow \alpha;\, \Gamma \vdash \Delta,\, \alpha.$$

Hence is the corresponding inference rule:

$$\frac{M;\, \Sigma,\, \phi^{+}(\alpha) \rightarrow \alpha;\, \Gamma \vdash \Delta,\, \alpha}{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \mu X.\, \phi^{+}(X)}\mu_R^{\alpha}.$$

Finally, the rules for eliminating the fixpoint operators on the left (in Γ) are dual to their right-hand side counterparts:

$$\frac{M;\,\Sigma,\,\alpha \to \phi^+(\alpha);\,\Gamma,\,\alpha \vdash \Delta}{M;\,\Sigma;\,\Gamma,\,\nu X.\,\phi^+(X) \vdash \Delta}\nu_L^\alpha$$

$$\frac{M;\,\Sigma;\,\Gamma,\,\psi \vdash \Delta \quad M' = (S,\,\to,\,S);\,\Sigma;\,\phi^+(\psi) \vdash \psi}{M;\,\Sigma;\,\Gamma,\,\mu X.\,\phi^+(X) \vdash \Delta}\mu_L.$$

Even though these rules are based solely on the Tarski's definitions of the fixpoints, one can give a practical intuition behind them. For instance, ν_R corresponds to an induction over time (or, more precisely, over the number of iterations of ϕ). The formula ψ is an "inductive invariant" which is true in the initial states (the base of the induction), and each iteration of ϕ preserves it. Therefore, $\phi(\psi)$ must hold globally on all states "reachable" by iterating ϕ. For a classical safety property $\mathbf{AG}\ \xi = \nu X.\,\xi \wedge \Box X$ this is precisely the induction over time: ξ holds at any initial state, and if there is an inductive invariant ψ that implies ξ and itself at the next time step, then ξ is also an invariant of the set of reachable states.

The intuition behind the μ_R^α rule is not as straightforward. For practical purposes, we may think of the least fixpoint as a *termination* formula. The formula $\phi(\alpha) \to \alpha$ is an "unwinding" of the fixpoint "backward" from the current state back to the initial state, and if we can reach an initial state this way, then we know our model "terminates."

This intuition leads to a more practical version of this rule that reduces the proof to a well-founded induction over natural numbers (arithmetic is included in the actual implementation, but the relevant rules are not discussed in this Chapter):

$$\frac{\begin{array}{ll}(1) & M;\,\Sigma,\,\phi(\alpha) \to \alpha;\,\Gamma \vdash \Delta,\,\psi(0)\\ (2) & \quad M';\,\Sigma,\,\phi(\alpha) \to \alpha;\,\psi(N) \vdash \Delta,\,\alpha\\ (3) & \qquad M';\,\Sigma,\,\phi(\alpha) \to \alpha;\,\cdot \vdash \Delta,\,(\forall 0 \le i < N).\,\psi(i) \to \phi^+(\psi(i+1))\end{array}}{M;\,\Sigma;\,\Gamma \vdash \Delta,\,\mu X.\,\phi^+(X)}\mu\mathrm{Ind}_R^\alpha,$$

where $M' = (S,\,\to,\,S)$. Here $\psi(i)$ is a μ-calculus formula that depends on a numeric parameter $i \in \mathcal{N}$, and N is some non-negative integer constant.

This new rule states that if there is a *measure* $\psi(i)$ on the reachable set of states that captures the set of states reachable in i iterations of ϕ from the set of initial states, and the index of the property of interest $\phi(\alpha)$ is N, then the least fixpoint of ϕ is larger than the set of initial states S. We will see how this rule is used in a simple example in Figure 1 on page 77.

Lemma 3.5.2. *The rule $\mu\mathrm{Ind}_R^\alpha$ is admissible in the proof system described above.*

Proof. To show the admissibility of the rule in our system, assume that its premisses (1), (2) and (3) have valid derivations. Starting from the fixpoint formula:

$$M;\, \Sigma;\, \Gamma \vdash \Delta,\, \mu X.\, \phi(X)$$

that we want to prove, we will build a proof of it from bottom up to the axioms. First, the μ_R^α rule adds an assumption $\phi(\alpha) \to \alpha$ to the invariant Σ and α to the conclusions Δ:

$$M;\, \Sigma,\, \phi(\alpha) \to \alpha;\, \Gamma \vdash \Delta,\, \alpha.$$

We then introduce a set of assumptions in Γ by repeated application of the Cut rule, resulting in a sequent valid by the simple propositional reasoning:

$$M;\, \Sigma,\, \phi(\alpha) \to \alpha;\, \Gamma,\, \psi(0),\, \psi(0) \to \phi^N(\psi(N)),\, \phi^N(\psi(N)) \to \phi^N(\alpha),\, \phi^N(\alpha) \to \alpha \vdash \Delta,\, \alpha.$$

The added assumptions have to be discharged as separate proof subgoals generated by the applications of the Cut rule:

$$M;\, \Sigma,\, \phi(\alpha) \to \alpha;\, \Gamma \vdash \Delta,\, \alpha,\, \psi(0) \tag{3.2}$$

$$M;\, \Sigma,\, \phi(\alpha) \to \alpha;\, \Gamma \vdash \Delta,\, \alpha,\, \psi(0) \to \phi^N(\psi(N)) \tag{3.3}$$

$$M;\, \Sigma,\, \phi(\alpha) \to \alpha;\, \Gamma \vdash \Delta,\, \alpha,\, \phi^N(\psi(N)) \to \phi^N(\alpha) \tag{3.4}$$

$$M;\, \Sigma,\, \phi(\alpha) \to \alpha;\, \Gamma \vdash \Delta,\, \alpha,\, \phi^N(\alpha) \to \alpha. \tag{3.5}$$

The subgoal (3.2) is proven by weakening it (rule w_R) to remove α on the right and match the assumption (1) of the $\mu\mathrm{Ind}_R^\alpha$ rule.

The next subgoal (3.3) is first "flattened" (rule $\to_R$):

$$M;\, \Sigma,\, \phi(\alpha) \to \alpha;\, \Gamma,\, \psi(0) \vdash \Delta,\, \alpha,\, \phi^N(\psi(N)),$$

then generalized to the set of *all* states of M (rule s_{Init}):

$$M' = (S,\, \to,\, S);\, \Sigma,\, \phi(\alpha) \to \alpha;\, \Gamma,\, \psi(0) \vdash \Delta,\, \alpha,\, \phi^N(\psi(N)),$$

split into N subgoals (again using Cut and propositional reasoning):

$$M';\, \Sigma,\, \phi(\alpha) \to \alpha;\, \Gamma,\, \phi^j(\psi(j)) \vdash \Delta,\, \alpha,\, \phi^{j+1}(\psi(j+1))$$

for $j \in \{0,\, \ldots,\, N-1\}$. The monotonicity rule Mono is applied j times to each of the j-th subgoal, reducing it to:

$$M';\, \Sigma,\, \phi(\alpha) \to \alpha;\, \Gamma,\, \psi(j) \vdash \Delta,\, \alpha,\, \phi(\psi(j+1)),$$

which is then proven by case splitting (rule Cut) on $(\forall 0 \leq i < N).\, \psi(i) \rightarrow \phi^{+}(\psi(i+1))$. The subgoal with this formula on the right is discharged by the assumption (3) of the $\mu\mathrm{Ind}^{\alpha}_{R}$ rule (after weakening). In the other subgoal:

$$M';\, \Sigma,\, \phi(\alpha) \rightarrow \alpha;\, \Gamma,\, \psi(j),\, (\forall 0 \leq i < N).\, \psi(i) \rightarrow \phi^{+}(\psi(i+1)) \vdash \Delta,\, \alpha,\, \phi(\psi(j+1))$$

the quantifier is instantiated with with j, and the sequent is proven by simple propositional reasoning.

The subgoal (3.4):

$$M;\, \Sigma,\, \phi(\alpha) \rightarrow \alpha;\, \Gamma \vdash \Delta,\, \alpha,\, \phi^{N}(\psi(N)) \rightarrow \phi^{N}(\alpha)$$

is reduced to

$$M';\, \Sigma,\, \phi(\alpha) \rightarrow \alpha;\, \Gamma,\, \psi(N) \vdash \Delta,\, \alpha,\, \alpha$$

by $\rightarrow_{R}$ and N applications of the Mono rule, and then is discharged by the assumption (2).

Finally, the subgoal (3.5):

$$M;\, \Sigma,\, \phi(\alpha) \rightarrow \alpha;\, \Gamma \vdash \Delta,\, \alpha,\, \phi^{N}(\alpha) \rightarrow \alpha$$

is, again, first "flattened" by $\rightarrow_{R}$:

$$M;\, \Sigma,\, \phi(\alpha) \rightarrow \alpha;\, \Gamma,\, \phi^{N}(\alpha) \vdash \Delta,\, \alpha,\, \alpha,$$

and then the Cut rule is applied to it N times introducing a series of assumptions

$$\phi^{N-1}(\alpha),\, \phi^{N-2}(\alpha),\, \ldots,\, \phi^{1}(\alpha),\, \alpha$$

into Γ, in this order, last one (α) proving the sequent by the Init rule. The corresponding subgoals that validate these assumptions (the second premisses of the Cut rule) are

$$M;\, \Sigma,\, \phi(\alpha) \rightarrow \alpha;\, \Gamma,\, \phi^{N}(\alpha),\, \ldots,\, \phi^{i+1}(\alpha) \vdash \Delta,\, \phi^{i}(\alpha)$$

for $i = \{0,\, \ldots,\, N-1\}$. Here the α on the right hand side is already removed (rule w_{R}). Applying the Mono rule i times yields

$$M;\, \Sigma,\, \phi(\alpha) \rightarrow \alpha;\, \Gamma,\, \phi(\alpha) \vdash \Delta,\, \alpha,$$

which is then proven by w_{I} (moving the invariant assumption $\phi(\alpha) \rightarrow \alpha$ into Γ) and propositional reasoning.

This completes the admissibility proof of $\mu\mathrm{Ind}^{\alpha}_{R}$. □

It is also easy to generalize this rule to an arbitrary well-founded induction over other domains, like ordinals or recursive datatypes, if we use $\mathrm{wf\_Ind}^{a}_{R}$ instead of Ind^{a}_{R} in the proof of admissibility.

Other Inference Rules. The rules above comprise the *basis* of the proof system. In practice, there is often a need for more powerful and specialized rules, and we will introduce some of them below and in the later chapters when we talk about concrete examples.

3.5.2 Cone of Influence

When the set of states in a model M is defined by means of *state variables*, and the transition relation is defined by *assignments* to those variables (e.g. as explained in Section 2.1.1), then it is often possible to determine which variables can have a potential influence of a given specification, and which cannot. The unimportant variables can, therefore, be removed from the model without changing the validity of the specification. We call such a reduction and the corresponding inference rule the *Cone of Influence*:

$$\frac{M';\,\Sigma;\,\Gamma\vdash\Delta}{M;\,\Sigma;\,\Gamma\vdash\Delta}\mathrm{COI},$$

where M' is the resulting reduced model.This reduction is very important in practical verification, and we explain it below more formally.

Let the model M be defined as $(S,\,\to,\,I)$ such that:

- $S = D_1 \times \cdots \times D_n$, where each D_i is the domain of a *state variable* v_i and there are n state variables in M; in other words, every state is a tuple $(d_1,\,\ldots,\,d_n)$ of values assigned to the corresponding state variables.

- Transition relation $\to\subseteq S \times S$ is defined as a set

  $$\{((x_1,\,\ldots,\,x_n),\,(x'_1,\,\ldots,\,x'_n)) \mid P(x_1,\,\ldots,\,x_n,\,x'_1,\,\ldots,\,x'_n)\},$$

  where the predicate P is $P(\vec{x},\,\vec{x}') = \bigwedge_{i=1}^{n} A_i(x'_i,\,\vec{x})$, and each formula A_i corresponds to an *assignment* to the *next state variable* x_i. That is, x_i will get some value d_i in the next state which satisfies $A_i(d_i,\,\vec{x})$ in the current state. If an assignment is deterministic ($v'_i := f_i(\vec{v})$, where f_i is a function), then A_i is simply an equality $x'_i = f_i(\vec{x})$. However, we allow A to be an arbitrary formula such that $\forall\vec{x}.\,\exists d.\,A(d,\,\vec{x})$; that is, there is always a next state value d for v_i, no matter what the values of the other variables are. Hence, an assignment can be nondeterministic.

  In addition, we assume that, in general, not all *current state* (i.e. unprimed) variables appear in A_i; we denote the set of unprimed variables that do appear in A_i by $\mathrm{COI}(A_i)$.

- $I(\vec{x})$ is any satisfiable predicate on the state variables.

Define a model $M|_V$ (the model M *restricted* to the set of variables V) to be

$$M|_V = (S|_V, \rightarrow|_V, I|_V),$$

where

- $S|_V = \bigotimes_{v_i \in V} D_i$ (a *projection* of S on the subspace defined by V)
- $\rightarrow|_V$ is a projection of $\rightarrow$ on $S|_V \times S|_V$, and
- $I|_V$ is a projection of I on $S|_V$.

The *Cone of Influence* of a state variable v_i is the smallest set of variables V_i that satisfies the following conditions:

- $\mathrm{COI}(A_i) \subseteq V_i$ (all the variables that v_i immediately depends on are included in its cone of influence);
- If $v_j \in V_i$, then $\mathrm{COI}(A_j) \subseteq V_i$ (that is, V_i is transitively closed under the variable dependency).

Finally, the Cone of Influence w.r.t. a formula is defined as follows. Let ϕ be a CTL formula, and V_ϕ be the set of state variables that appear in this formula. Then

$$\mathrm{COI}(\phi) = \bigcup_{v \in V_\phi} \mathrm{COI}(v).$$

This definition extends to a set of formulas Φ as follows:

$$\mathrm{COI}(\Phi) = \bigcup_{\phi \in \Phi} \mathrm{COI}(\phi).$$

From the definition of Cone of Influence for a single variable v, it is obvious that the variables outside of $\mathrm{COI}(v)$ cannot have any effect on the value of v, and therefore, the behavior of the model M w.r.t. v will be exactly the same as that of $M|_{\mathrm{COI}(v)}$. In fact, $M|_{\mathrm{COI}(v)}$ is *bisimilar* to M w.r.t. v,[3] as defined below. The generalization of this fact to multiple variables is straightforward.

[3]More precisely, w.r.t. an equivalence relation H that equates the states of the two models where the values of the variable v are the same.

Definition 3.5.3. Given two models, or Kripke structures, $M_1 = (S_1, \rightarrow_1, I_1)$ and $M_2 = (S_2, \rightarrow_2, I_2)$, a binary relation $R \subseteq S_1 \times S_2$ is called a *bisimulation relation* w.r.t. another (given) relation $H \subseteq S_1 \times S_2$ if the following conditions hold:

- $R \subseteq H$, that is, if $R(s_1, s_2)$ holds, then necessarily $H(s_1, s_2)$ holds;
- For any $s_1, q_1 \in S_1$ such that $s_1 \rightarrow_1 q_1$, and for any $s_2 \in S_2$ such that $R(s_1, s_2)$, there exists $q_2 \in S_2$ such that $s_2 \rightarrow_2 q_2$ and $R(q_1, q_2)$;
- For any $s_2, q_2 \in S_2$ such that $s_2 \rightarrow_2 q_2$, and for any $s_1 \in S_1$ such that $R(s_1, s_2)$, there exists $q_1 \in S_1$ such that $s_1 \rightarrow_1 q_1$and $R(q_1, q_2)$.

Two models M_1 and M_2 are called *bisimilar* (denoted $M_1 \sim M_2$) if there exists a bisimulation relation R such that:

- For any $s_1 \in I_1$ there exists $s_2 \in I_2$ such that $R(s_1, s_2)$, and
- For and $s_2 \in I_2$ there exists $s_1 \in I_1$ such that $R(s_1, s_2)$.

The definition of bisimulation was first introduced by Park in [Par81]. Then Hennessy and Milner [HM85] defined a modal logic which is the same as the propositional fragment of our μ-calculus, only without the fixpoints, and showed that two models M_1 and M_2 are bisimilar if and only if they satisfy exactly the same set of Hennessy-Milner logic formulas.

The same result carries over to our version of the first-order modal μ-calculus, and therefore, we can say that a formula ϕ holds in all the initial states of M if and only if it holds in all the initial states of $M|_{\mathrm{COI}(\phi)}$.

The definition of the cone of influence can also be extended to models with more complex assignments, or with other types of assignments (e.g. *combinational assignments*).

3.5.3 Conservative Abstraction

If all the formulas in a sequent are "universal," that is, only $\forall$ quantifiers and $\Box$ modality is used in positive subformulas, and their duals in negative ones, then we can apply a *conservative abstraction* to the model:

$$\frac{M';\,\Sigma;\,\Gamma \vdash \Delta}{M;\,\Sigma;\,\Gamma \vdash \Delta}\mathrm{Abs},$$

where $M' = (S', \rightarrow', I')$ is an abstraction of $M = (S, \rightarrow, I)$. We do not specify the exact details of how this abstraction is computed, and leave it to the actual implementation of the proof system. The only restriction is that application of abstraction must be sound.

One of the most common types of abstractions is *existential abstraction*, where the resulting model is a *simulation* of the original model. That is:

$$M \preceq M',$$

where the relation $\preceq$ is defined as follows.

Let $\mathcal{L}$ be the set of all atomic subformulas in Σ, Γ, and Δ, and $L : S \to 2^{\mathcal{L}}$ be a *labeling function* that maps each state to the set of *labels* (atomic formulas) that hold in that state. Define $H \subseteq S \times S'$ to be a relation between the states from the concrete and the abstract models such that $H(s,\, s')$ iff $L(s) = L(s')$.

Definition 3.5.4. A binary relation $R \subseteq S \times S'$ is called a *simulation relation* w.r.t. H if the following conditions hold:

- $R \subseteq H$, that is, if $R(s,\, s')$ holds, then necessarily $H(s,\, s')$ holds;
- For any $s,\, q \in S$ such that $s \to q$, and for any $s' \in S'$ such that $R(s,\, s')$, there exists $q' \in S'$ such that $s' \to' q'$ and $R(q,\, q')$;

We say that M *is simulated by* M' (denote this by $M \preceq M'$) if there exists a simulation relation $R \subseteq S \times S'$ w.r.t. H such that:

- For any $s \in I$ there exists $s' \in I'$ such that $R(s,\, s')$.

Intuitively, the abstract model can mimic (or *simulate*) any behavior that the original model M can exhibit. Therefore, if we are only interested in the "universal" properties of M, that is, those properties that require every execution trace of M satisfy some other "universal" property, it is sufficient to verify that the abstraction M' of M satisfies this property, since we will prove it for all traces in M', and therefore, all the traces of M (which are also traces of M') will satisfy it.

It is rare in practice that both concrete and abstract models are given in advance, and the problem is only to check that the abstract model indeed simulates the concrete one. Most often, only the concrete model is provided, and the abstract model needs to be built. We use the so-called *abstraction function* approach to build a *correct by construction* abstract model. Intuitively, given a concrete model, one only needs to come up with a relatively simple *abstraction function* which defines the abstract model. If this abstraction function satisfies certain properties, then the resulting abstract model will indeed be a simulation of the original concrete one.

Definition 3.5.5. A function $h : S \to S'$ is called an *abstraction function* of $M = (S, \to, I)$, if $H(s, h(s))$ holds for any state $s \in S$, where H is as in Definition 3.5.4.

An abstraction function h defines an abstract model $M' = (S', \to', I')$ as follows:

- $S' = h(S)$; that is, the set of abstract states is the range of h;
- $\to' \supseteq \{(h(s), h(s')) \mid \text{for all } s \text{ and } s' \text{ such that } s \to s'\}$; and
- $I' \supseteq \{h(s) \mid s \in I\}$.

It is fairly straightforward to check that $M \preceq M'$. Notice, that the transition relation $\to'$ can be an arbitrary superset of the relation defined by the abstraction function, and the same is true for the set of initial states I'.

Since we are interested in using abstraction for practical examples, and both the model and the properties in those examples are often defined in terms of state variables, it is convenient to be able to express the abstraction function h in terms of state variables as well.

Assume that the original model $M = (S, \to, I)$ is defined as in Section 3.5.2, namely:

- $S = D_1 \times \cdots \times D_n$, where D_i is the domain of the i-th state variable v_i (and there are n state variables total);
- $\to \subseteq S \times S$ is a transition relation defined with possibly nondeterministic next state assignments to each state variable:

$$\to = \{(\vec{v}, \vec{v}') \mid \bigwedge_{i=1}^{n} A_i(v_i', \vec{v})\};$$

- $I \subseteq S$ is a non-empty initial state predicate defined as a logical formula (without temporal operators) over the state variables.

Suppose, for simplicity, that the current formulas in Σ, Γ, and Δ (or *specifications*) contain only equalities of the form $v = c$ and $v_1 = v_2$, where v's are state variables, and c is a constant. This restriction is by no means necessary, and is only here for illustration purposes.

One of the simple and intuitive ways to define an abstraction function h is to *eliminate* some state variables, or replace all of their instances in M with completely nondeterministic choices and remove them from the state. This reduction is often called *existential* *abstraction*, and we say that the removed variables are *abstracted*. Again, for simplicity, assume that only one variable v_1 is abstracted. The abstract model M', therefore, has the set of states $S' = D_2 \times \cdots \times D_n$ (so, it is just like S, only D_1 is omitted). If the abstracted variable v_1 does not occur in the specifications, then the following function $h : S \to S'$ is obviously an abstraction function:

$$h(v_1, v_2, \ldots, v_n) = (v_2, \ldots, v_n).$$

The reason for this is that the labeling (that is, the validity of the atomic formulas) in any state s does not depend on v_1; thus, $L(s) = L(h(s))$ for any s. Moreover, the abstract model M' induced by such h can be obtained from the original M with simple syntactic transformations. Namely, each assignment formula $A_i(v_i', \vec{v})$ for $i \geq 2$ is rewritten as $\exists v_1 .\, A_i(v_i', \vec{v})$, and then simplified. For example, if v_1 only occurs in $A_i(v_i', \vec{v})$ as $v_1 = e$ (where e is some expression), then define the new assignment A_i' of the abstract model as follows:

$$A_i'(v_i', \vec{v}) \equiv \exists v_1 .\, A_i(v_i', \vec{v}) \equiv A_i(v_i', \vec{v})[\mathbf{true}/v_1 = e] \vee A_i(v_i', \vec{v})[\mathbf{false}/v_1 = e].$$

That is, the entire equality $v_1 = e$ can be replaced by either $\mathbf{true}$ or $\mathbf{false}$, nondeterministically. In general, one can use *abstract interpretation* of interpreted functions to generate such nondeterministic choices, but it is out of the scope of this brief illustration.

The problem with the above construction of h is that it is often desirable to abstract variables occurring in the specification, and doing so can make h violate the property of being an abstraction function. The reason for this is quite simple: an atomic formula $v_1 = e$ cannot be definitely evaluated in any abstract state; it can be true or false nondeterministically, and therefore, the entire specification containing it may become nondeterministic, therefore, the function h may not preserve the labeling.

Fortunately, there is a simple way to correct this problem: for any atomic formula containing v_1, introduce a new Boolean state variable b with no constraints on the next state transition (i.e, it is completely nondeterministic), and replace this atomic formula by b. The abstract state space, therefore, may contain more state variables that the concrete one, but the ranges of the additional variables will be finite and small, whereas the range of the abstracted variables may be very large or even infinite.

Again, for simplicity, assume that only one such new Boolean variable b was introduced for an atomic formula $v_1 = e$. The "corrected" abstraction function $h' : S \to \{0, 1\} \times S'$ is then defined as follows:

$$h'(v_1, v_2, \ldots, v_n) = ([\mathbf{if}\ v_1 = e\ \mathbf{then}\ \mathbf{true}\ \mathbf{else}\ \mathbf{false}],\ v_2, \ldots, v_n).$$

The first component of $h'(\vec{v})$ defines the value of the new Boolean variable in the abstract state. The transition relation of M' is defined the same way as for the original h using the syntactic transformation to the assignments, and the new Boolean state variable b is left completely unconstrained. Namely, the abstract assignment A'_i for each i-th variable in the abstract model is defined as follows:

$$A'_i(v'_i, \vec{v}) \equiv A_i(v'_i, \vec{v})[b/v_1 = e].$$

It is not hard to check that such an abstract model M' is indeed induced by the function h'. The fact that h' is an abstraction function for M follows from the construction of h' and the fact that we label an abstract state where b is true by $v_1 = e$, and therefore, h' preserves the labeling for all states.

This abstraction procedure can be generalized to handle more complex atomic formulas and finer abstraction of variables. Consider, for example, a formula of the form:

$$\forall x. \, (v_1 = x \to \phi(\vec{v}, x))$$

that we want to check on the model M:

$$M; \cdot; \cdot \vdash \forall x. \, (v_1 = x \to \phi(\vec{v}, x)).$$

After eliminating the quantifier (rule $\forall_R$) and "flattening" the sequent we obtain

$$M; \cdot; v_1 = a \vdash \phi(\vec{v}, a),$$

where a is a new Skolem constant. Suppose that the validity of ϕ depends only of the fact that $v_1 = a$, but the concrete value of a does not matter. If the domain D_1 of the variable v_1 is the only very large or infinite variable domain in M, and the rest of the model is small enough, then the proof of this sequent can be easily finished by model checking (the MC rule) if we can find a suitable abstraction for v_1. However, the simple existential abstraction of v_1 is too coarse in this case, since we need to retain some information about v_1, namely that $v_1 = a$. Introducing a new Boolean variable b for $v_1 = a$ may not solve the problem, since if ϕ contains another equality, say, $v_2 = a$, its relationship to the new variable b will be lost in the abstraction and we will no longer be able to infer that $v_2 = v_1$.

A solution to this is *to abstract the domain* D_1. For simplicity we assume that the domains of comparable variables are always equal. That is, if it makes sense to write $v_1 = v_2$, then the corresponding domains must be the same: $D_1 = D_2$. In practice, this restriction corresponds to the requirement that all the expressions and formulas are *well-typed*, which is quite reasonable in real design languages.

If v_2 is the only variable that can be compared with v_1 in M and ϕ, then the abstract model M' is constructed from M by replacing D_1 and D_2 in the state space S by the set $D' = \{a, \perp\}$:

$$S' = \{a, \perp\} \times \{a, \perp\} \times D_3 \times \cdots \times D_n,$$

and supplying the *abstract interpretation* of the equality operator '$=$' on the new abstract domain D' as follows:

$$(x = y) := \begin{cases} \mathbf{true}, & \text{if } x \doteq a \wedge y \doteq a \\ \mathbf{false}, & \text{if } (x \doteq a \wedge y \doteq \perp) \vee (x \doteq \perp \wedge y \doteq a) \\ \mathbf{true} \text{ or } \mathbf{false} \text{ nondeterministically}, & \text{if } x \doteq \perp \wedge y \doteq \perp, \end{cases}$$

where $\doteq$ is the "true" equality in its standard interpretation. Intuitively, the special symbol $\perp$ represents all the elements from the original D_1 that are different from a, and therefore, when we compare two $\perp$ values with each other in the abstract model, the exact outcome cannot be determined.

The actual abstraction function h is again constructed as a syntactic transformation on the model M mapping all the constants from D_1 and D_2 into D' and rewriting all the equalities w.r.t. the above abstract interpretation. For instance, a possible implementation of h would replace an expression $v_1 = v_2$ anywhere in M by

$$\begin{array}{lll} \mathbf{if}\ v_1 = a \wedge v_2 = a & \mathbf{then} & \mathbf{true} \\ \mathbf{elsif}\ (v_1 = a \wedge v_2 = \perp) \vee (v_1 = \perp \wedge v_2 = a) & \mathbf{then} & \mathbf{false} \\ & \mathbf{else} & b \end{array}$$

where b is a new unconstrained Boolean state variable. Notice that the constant a in the abstract model is now an *interpreted* constant, hence, the connection with the original possibly infinite domain D_1 is completely eliminated.

This abstraction preserves more information about state variables' equalities, and therefore, has a higher chance of preserving more involved properties like the one we have mentioned above. The possibly infinite domains D_1 and D_2 in the resulting abstract model are replaced with small finite domains, and if the rest of the model is sufficiently small, then it can be verified using model checking techniques.

However, this abstraction is not directly applicable when the formula has more than one uninterpreted (Skolem) constant from the same domain. To illustrate the problem, suppose that we have two uninterpreted constants a_1 and a_2 in our model M and the formula ϕ. According to the semantics of the sequent, such a model would satisfy ϕ if it satisfies ϕ for *all possible interpretations* of a_1 and a_2. Assume again, for simplicity, that both M

and ϕ only depend on whether $a_1 = a_2$ or not. Then there are two important classes of interpretations that matter: the one where $a_1 = a_2$, and the one where $a_1 \neq a_2$. Hence, we have to consider two abstractions, the one that maps the domain D of a_1 and a_2 to $\{a, \perp\}$ and interprets both a_1 and a_2 with the same interpreted constant a, and the other abstraction mapping D to $\{a_1, a_2, \perp\}$, where a_1 and a_2 are now *distinct* interpreted constants.

In general, for n uninterpreted constants from the same domain, $a_1, \ldots, a_n$, we have to consider all possible *equivalence classes* over the set $\{a_1, \ldots, a_n\}$, and generate the corresponding number of different abstractions. This leads to a new specialized inference rule:

$$\frac{M_{\sigma_1};\, \Sigma_{\sigma_1};\, \Gamma_{\sigma_1} \vdash \Delta_{\sigma_1} \quad \cdots \quad M_{\sigma_N};\, \Sigma_{\sigma_N};\, \Gamma_{\sigma_N} \vdash \Delta_{\sigma_N}}{M;\, \Sigma;\, \Gamma \vdash \Delta}\text{abstractSplit}.$$

In this rule, we collect all the Skolem constants used in the original sequent, both in the model and in the formulas, and construct all different sets of assumptions σ_i about the relationships among these Skolem constants (N is the number of different σ_i's). These assumptions, in the simplest case, are equalities or dis-equalities between each pair of Skolem constants. This rule is precise (or *invertible*) when Skolem constants are only compared for equality among each other or used as array indices. This condition could be easily weakened to allow Skolem constants be compared with some concrete constants (e.g., if a term $a = 0$ appears in a formula), and even used in some predicates other than the equality, but this is beyond the scope if this example. For our purposes it is sufficient to view the set of assumptions σ_i as an *equality relation* over Skolem constants defined by its *equivalence classes*: $\sigma_i = \{S^i_1, \ldots, S^i_{k_i}\}$. For each equivalence relation σ_i we modify the model appropriately (i.e. merge all the array state variables $x(a_1)$ and $x(a_2)$ such that a_1 and a_2 are in the same equivalence class of σ_i) and evaluate all the predicates of the form $a_1 = a_2$ to either $\mathtt{true}$ or $\mathbf{false}$, depending on whether a_1 and a_2 are in the same equivalence class or not. After this transformations for each σ_i, we obtain new sequents in which different Skolem constants are guaranteed to have only different interpretations, and can be replaced with *abstract constants* (so that further rules would not confuse them with unreduced Skolem constants). Thus, abstract constants are essentially Skolem constants, except that different abstract constants are guaranteed to have different interpretations, and $b_1 = b_2$ iff b_1 and b_2 are one and the same constant.

3.6 Simple Examples

For a better intuition behind the inference rules and the way they are supposed to be used, we provide a simple example and its complete proof.

3.6.1 Liveness: an Unbounded Counter

We have seen how we can use the proof system to prove a simple safety property, which essentially boils down to an induction over time. For a liveness property, we need to show that something good will eventually happen in finite number of steps, and therefore, we expect to have a proof resembling a proof of termination. One way of proving termination is to find a well-founded measure on a program state that strictly decreases over time and is bounded from below. We shall see how this idea is realized in our proof system from the next simple example.

Consider the program (in *SyMP*) in Algorithm 1.

Algorithm 1 Unbounded counter.

```
MODULE Counter =
begin
  StateVar c: int
  init(c) := 0;
  next(c) := c+1;

  THEOREM progress = self ⊨ ∀x. x ≥ 0 → AF c = x
end
```

First, let us restate the `progress` theorem in the μ-calculus:

Theorem 3.6.1.

$$M = (S, \to, \{c = 0\}) \vdash \forall x.\, \mu X.\, c = x \vee \Box X.$$

We prove this theorem by using a measure $\psi(i) \equiv i \leq c \leq x$ in the rule $\mu\mathrm{Ind}_R^\alpha$. The rule makes it possible to convert the general liveness problem into 3 simple propositional formulas in Presburger arithmetic. To perform this conversion and complete the proofs of Presburger formulas, we introduce two new inference rules. The first one uses the information from the model to instantiate state variables and eliminate some of the $\Box$ and $\diamond$ modalities:

$$\frac{M;\, \Sigma;\, \Gamma' \vdash \Delta'}{M;\, \Sigma;\, \Gamma \vdash \Delta}\mathrm{subst\_asst}.$$

Intuitively, all the state variables that occur in the formulas and are not under the scope of any fixpoint operator or a modality are replaced with their initialization expressions from the model. Likewise, the variables that are in the scope of just one modality and have a deterministic `next` assignment in the model are replaced by the right hand side expression of their `next` assignment. Effectively, a part of the model is substituted into the formulas from Γ and Δ yielding Γ' and Δ' respectively.

More formally, the following transformations are applied to every formula $\phi \in \Gamma \cup \Delta$ and every state variable v that occurs in ϕ:

- If the initial value of v (the right hand side expression e_{Init} of its `init` assignment) is deterministic, then every occurrence of v in ϕ that is not under the scope of a fixpoint operator or a modality is replaced by e_{Init}.
- Let ψ be a formula without modalities and fixpoints such that $\Box\psi$ or $\Diamond\psi$ is a subformula of ϕ and it is not under the scope of any other modality or a fixpoint. If the next state of every state variable v that occur in ψ is defined by the `next`-assignment with a deterministic right hand side expression e_v, then any occurrence of v in ψ is replaced by e_v. After all the state variables are replaced in a particular occurrence of ψ in ϕ, the corresponding $\Box$ or $\Diamond$ modality is removed.
- The rest of the formula ϕ remains unchanged.

Applying these transformations to all formulas in Γ and Δ generates the new Γ' and Δ' respectively in the premiss of the subst_asst rule. This rule is not derivable in the original proof system because the notion of a model is left open to concrete implementations, and therefore, there is no general enough way to transfer this kind of information from the model to the formulas. In this example, a concrete way of defining the model as a set of assignments is used, and therefore, the subst_asst rule can now be formulated for this particular type of model representation.

The second rule implements a decision procedure for quantifier-free Presburger arithmetic:

$$\frac{}{M;\, \Sigma;\, \Gamma,\, \Gamma' \vdash \Delta,\, \Delta'}\mathrm{Presburger},$$

where $\bigwedge \Gamma' \to \bigvee \Delta'$ is a generally valid quantifier-free formula in Presburger arithmetic. The state variables and any uninterpreted symbols of the integer type (Skolem constants, uninterpreted module parameters, etc.) are considered variables for the purpose of the decision procedure.

Proof.

1. $M;\ \cdot;\ \cdot \vdash \forall x.\, x \geq 0 \to \mu X.\, c = x \vee \Box X$ ($\forall^a_R$, $\to_R$, 2)
   Eliminate the quantifier (introducing a new Skolem constant a) and "flatten" the sequent.

2. $M;\ \cdot;\ a \geq 0 \vdash \mu X.\, c = a \vee \Box X$ ($\mu\mathrm{Ind}^{\alpha}_R$, 3, 4, 5)
   Apply the inductive rule for eliminating μ-fixpoint; this generates 3 subgoals.

3. $M;\ \Sigma;\ a \geq 0 \vdash 0 \leq c \leq a$ (subst_asst, Init), where $\Sigma = \{(c = a \vee \Box\alpha) \to \alpha\}$.
   Instantiate the initial value of the state variable c (which is $c = 0$) and discharge this simple formula with the Init rule.

4. $M';\ \Sigma;\ a \leq c \leq a \vdash c = a \vee \Box\alpha$ ($\vee_R$, $\mathrm{Presburger}$)
   The sequent contains a generally valid system of Presburger formulas, "flatten" and discharge it.

5. $M' = (S, \to, S);\ \Sigma;\ \cdot \vdash (\forall 0 \leq i < a).\, i \leq c \leq a \to (c = a \vee \Box(i+1 \leq c \leq a))$ ($\forall^b_R$, 6)
   The inductive step: first, eliminate the quantifier; this introduced a new Skolem constant b.

6. $M' = (S, \to, S);\ \Sigma;\ \cdot \vdash b \leq c \leq a \to (c = a \vee \Box(b+1 \leq c \leq a))$ ($\mathrm{subst\_asst}_R$, 7)
   Then instantiate the next-state values of c into the formula (the initial assignments are removed from M' and are not instantiated).

7. $M' = (S, \to, S);\ \Sigma;\ \cdot \vdash b \leq c \leq a \to (c = a \vee (b+1 \leq c+1 \leq a))$ ($\mathrm{Presburger}$)
   Finally, discharge the resulting valid Presburger formula.

$\Box$

The proof proceeds by first eliminating the universal quantifier and the top-level implication, introducing a new Skolem constant a for the quantified variable x (step 1). Then the inductive form of the μ-elimination rule is applied (step 2), yielding three new subgoals. The base case (step 3) is proven by instantiating the initial value of c from the model, which simplifies the formula in Δ to $0 \leq a$, and it is taken care of by the Init rule. The "termination" condition (step 4) is proven by the direct application of Presburger decision procedure after separating the irrelevant $\Box\alpha$ component. And, finally, the inductive step 6 is discharged by replacing the value of c in the scope of the $\Box$ operator by the expression from its `next` assignment, which deterministically depends on `c` at the current time. This effectively eliminates the $\Box$ modality. Since the initial state assignments are pruned from the model at this point, the instances of the variable `c` outside of the $\Box$ operator are left as they are. The formula then becomes a propositional Presburger formula (step 7) which is proven by the corresponding decision procedure.

3.7 Restriction to Linear Time μ-Calculus

In the previous sections we have described a proof system for the traditional branching time μ-calculus. The proof system closely follows Kozen's proof system [Koz83] which has been proven complete by Walukiewicz [Wal95]. This suggests that our proof system might also be complete in the sense that it can prove any formula which is valid in any model (assuming we do not have any interpreted symbols like equality or arithmetic operators). Since μ-calculus is more expressive than CTL*, theoretically what we have is already sufficient for any properties that arise in most of the verification needs. However, in practice this generality may come at too high a price. Consider, for example, a sequent

$$M;\ \cdot;\ \mathbf{AG}\ \phi_1 \vdash \mathbf{AG}\ \phi_2,$$

and we know that $\phi_1 \to \phi_2$ holds globally in the model. Moreover, this implication can be easily model checked after a simple abstraction. Unfortunately, we cannot apply the Abs rule directly to this sequent since Γ contains a formula $\mathbf{AG}\ \phi_1$, and therefore, the sequent is not "universal." In the current system we have to eliminate the fixpoints, find the appropriate invariants, and, maybe, do some tedious theorem proving before we reduce the problem to proving $\phi_1 \to \phi_2$.

However, if our model is "linear", that is, consists of one single path, then existential path quantifiers can be safely converted into universal ones and we can apply the Abs rule to the original formula. This suggests that a *linear time* variant of the first order μ-calculus is more appropriate in such cases.

3.7.1 Syntax

The syntax of the first order linear time μ-calculus is exactly the same as the branching time one, except that we have only one modality $\circ$ instead of two ($\Box$ and $\Diamond$). To summarize, the full syntax is the following.

$$\begin{array}{rcl}
\text{Term} & ::= & c \,|\, x \,|\, v \,|\, f(t_1,\ldots,t_n) \\
\text{Predicate} & ::= & X \,|\, P(t_1,\ldots,t_n) \\
\text{Formula} & ::= & \text{Predicate} \,|\, \phi_1 \wedge \phi_2 \,|\, \phi_1 \vee \phi_2 \,|\, \neg\phi \,|\, \phi_1 \rightarrow \phi_2 \\
 & & |\ \circ\,\phi \\
 & & |\, \forall x.\,\phi \,|\, \exists x.\,\phi \\
 & & |\, \mu X.\,\phi^+(X) \,|\, \nu X.\,\phi^+(X)
\end{array}$$

3.7.2 Semantics

Linear time formulas are interpreted by the set of *paths* π in the model M. We denote by $[\![\phi]\!]^{\mathrm{p}}_M e$ the set of paths that satisfy the formula ϕ. Here e is an environment that, as before, assigns an interpretation to propositional and object variables in ϕ, with the only difference that propositional variables are assigned sets of paths rather than sets of states. For a path $\pi = s_1 s_2 \ldots$ we write π_i to denote the suffix of π starting at i-th state: $\pi_i = s_i s_{i+1} \ldots$, and $\pi(i)$ stands for the i-th state of the path: $\pi(i) = s_i$. The set of all suffixes of a path π is $\mathrm{suff}(\pi)$:

$$\mathrm{suff}(\pi) = \{\pi_i \,|\, i \in \mathcal{N}\}.$$

Formally, we define the semantics of a linear-time μ-calculus formula ϕ as follows:

$$[\![\phi]\!]^{\mathrm{p}}_M e = \bigcup_{\pi \in \mathrm{paths}(M)} [\![\phi]\!]^{\pi}_M e,$$

where $[\![\phi]\!]^{\pi}_M e$ is the set of all *suffixes* of π that satisfy ϕ. It is defined in exactly the same way as in Section 3.4.2 except for predicates, the modality, and the fixpoints. As before, we often omit the subscript M when the model is unambiguous. A predicate P is interpreted by the set of paths that start in the states where this predicate is true:

$$[\![P(t_1,\ldots,t_n)]\!]^{\pi} e = \{\pi \,|\, \pi(0) \in [\![P(t_1,\ldots,t_n)]\!] e\}.$$

The semantics for the other connectives is as follows:

$$
\begin{array}{rcl}
[\![X]\!]^{\pi} e & = & e(X) \\
[\![\phi_1 \wedge \phi_2]\!]^{\pi} e & = & [\![\phi_1]\!]^{\pi} e \cap [\![\phi_2]\!]^{\pi} e \\
[\![\phi_1 \vee \phi_2]\!]^{\pi} e & = & [\![\phi_1]\!]^{\pi} e \cup [\![\phi_2]\!]^{\pi} e \\
[\![\neg\phi]\!]^{\pi} e & = & \overline{[\![\phi]\!]^{\pi} e} \\
[\![\phi_1 \to \phi_2]\!]^{\pi} e & = & \overline{[\![\phi_1]\!]^{\pi} e} \cup [\![\phi_2]\!]^{\pi} e \\
[\![\circ\phi]\!]^{\pi} e & = & \{\pi_i \mid \pi_{i+1} \in [\![\phi]\!]^{\pi} e\} \\
[\![\forall x.\, \phi]\!]^{\pi} e & = & \bigcap_{a \in D_x} [\![\phi]\!]^{\pi} e[x \leftarrow a] \\
[\![\exists x.\, \phi]\!]^{\pi} e & = & \bigcup_{a \in D_x} [\![\phi]\!]^{\pi} e[x \leftarrow a] \\
[\![\mu X.\, \phi(X)]\!]^{\pi} e & = & \bigcap \{S \subseteq \mathrm{suff}(\pi) \mid [\![\phi(X)]\!]^{\pi} e[X \leftarrow S] \subseteq S\} \\
[\![\nu X.\, \phi(X)]\!]^{\pi} e & = & \bigcup \{S \subseteq \mathrm{suff}(\pi) \mid S \subseteq [\![\phi(X)]\!]^{\pi} e[X \leftarrow S]\}.
\end{array}
$$

The entailment $M,\, \pi \models_p \phi$ stands for "$\pi(0) \in I_M$ implies $\pi \in [\![\phi]\!]^{\mathrm{p}}_{M}$.", where I_M is the set of initial states of M, and $M \models_p \phi$ means $M,\, \pi \models_p \phi$ holds for all π in M.

3.7.3 Inference Rules

The sequent for the proof system changes its meaning accordingly:

$$M,\, \Sigma,\, \Gamma \vdash_p \Delta$$

means that for any path π of M starting from one of the states in I, if $M,\, \pi_i \models \bigwedge \Sigma$ for any i (that is, any suffix of π satisfies all the formulas in Σ), and $M,\, \pi \models \bigwedge \Gamma$, then $M,\, \pi \models \bigvee \Delta$.

Notice, that the semantics $[\![\phi]\!]^{\pi} e$ essentially restricts our model M to a fixed path π, and within this path π it corresponds one-to-one to the original branching time semantics. Thus, we can think of the new sequent $M,\, \Sigma,\, \Gamma \vdash_p \Delta$ as a family of instances of the original sequent $M|_{\pi};\, \Sigma;\, \Gamma \vdash \Delta$ for all π starting from a state in I. Given this observation, it is clear that most of the inference rules in the new proof system are exactly the same as in the original one. We only have to specialize the rules for modalities to the new modality $\circ$. If a model has only one single path, the $\Box$ and $\diamond$ modalities become the same, and we simply rewrite all the rules with this fact in mind.

$$
\frac{M,\, \Sigma,\, \Gamma \vdash_p \Delta,\, \circ\phi_1 \wedge \circ\phi_2}{M,\, \Sigma,\, \Gamma \vdash_p \Delta,\, \circ(\phi_1 \wedge \phi_2)}\ \circ\wedge_R
\qquad
\frac{M,\, \Sigma,\, \Gamma,\, \circ\phi_1 \wedge \circ\phi_2 \vdash_p \Delta}{M,\, \Sigma,\, \Gamma,\, \circ(\phi_1 \wedge \phi_2) \vdash_p \Delta}\ \circ\wedge_L
$$

$$
\frac{M,\, \Sigma,\, \Gamma \vdash_p \Delta,\, \circ\phi_1 \vee \circ\phi_2}{M,\, \Sigma,\, \Gamma \vdash_p \Delta,\, \circ(\phi_1 \vee \phi_2)}\ \circ\vee_R
\qquad
\frac{M,\, \Sigma,\, \Gamma,\, \circ\phi_1 \vee \circ\phi_2 \vdash_p \Delta}{M,\, \Sigma,\, \Gamma,\, \circ(\phi_1 \vee \phi_2) \vdash_p \Delta}\ \circ\vee_L
$$

$$\frac{M,\,\Sigma,\,\Gamma,\,\circ\phi \vdash_p \Delta}{M,\,\Sigma,\,\Gamma \vdash_p \Delta,\,\circ\neg\phi}\neg\circ_R \qquad \frac{M,\,\Sigma,\,\Gamma \vdash_p \Delta,\,\circ\phi}{M,\,\Sigma,\,\Gamma,\,\circ\neg\phi \vdash_p \Delta}\neg\circ_L$$

The rules for the relationship between modalities are irrelevant in the linear time logic (there is only one modality here), and we skip those.

Although the MC rule does not change its form, the formula that we run a model checker on is a *linear time* formula, and we have to do linear time model checking. And finally, the rule for abstraction Abs loses its restrictions (since the linear time logic is preserved by the conservative abstraction without any restrictions, unlike in the branching time case) and can be applied to any sequent at any point in the proof. This gain is the entire purpose of the restriction to the linear time logic, and we will see in the next example how it simplifies the verification. Intuitively, since the abstraction can be applied earlier, we can use model checking engine earlier, thus making easier the theorem proving part. This, of course, comes with a price: linear time model checking is a substantially harder problem than branching time one.

3.8 Circular Reasoning

In hardware verification it is often the case that some components depend on others and *vice versa*. For instance, the data on the write-back bus in Tomasulo's OOO algorithm depends on the data stored in reservation stations, and reservation stations in turn fetch their data from the bus. Therefore, in order to prove the correctness of the data on the bus we need to assume it is correct in reservation stations and vice versa. One way of breaking this circularity is by induction over time. We assume the correctness of data in the reservation stations at time $t-1$ and prove the bus correct at time t. Then assume it is correct at time $t-1$ and prove the correctness of reservation stations at time t. In more complex situations we may have to assume the correctness of certain parts at time t and still maintain soundness if the "cyclic" dependencies are always broken by at least one cycle of time delay.

More formally, this criterion can be formulated as follows. Let $G = (V,\,E)$ be a directed graph, where vertices V are formulas, and there is an edge $E(\phi_1,\,\phi_2)$ between two formulas $\phi_1,\,\phi_2 \in V$ if ϕ_1 is used as an assumption to prove ϕ_2. Additionally, we mark the edge $E(\phi_1,\,\phi_2)$ as having a *time delay* if ϕ_1 is assumed at or before time $t-1$ when proving ϕ at time t. A proof by "circular" reasoning is sound if any cycle in G contains an edge marked by the time delay.

For instance, if the bus in our model propagates values instantaneously, then we have to assume the correctness of the bus up to time t in proving the correctness of reservation stations at time t. This is still sound, since the bus depends on reservation stations only up to time $t-1$.

In our proof system both right rules for the fixpoints ν_R and $\mu\mathrm{Ind}_R^\alpha$ have a subgoal of the form $\psi \to \phi^+(\psi)$. If the invariant ψ has the form $\psi \equiv \bigwedge_{i\in A}\psi_i$ (here A is a finite set of indices), then we can prove N subgoals (indexed by l) of the form

$$\left[\bigwedge_{i\in A_l^0}\psi_i \wedge \bigwedge_{n=1}^{H_l}\phi^+(\bigwedge_{j\in A_l^n}\psi_j)\right] \to \phi^+(\bigwedge_{k\in A'_l}\psi_k),$$

such that $A_l^n \subseteq A$ and $A'_l \subseteq A$, and for any $0 \le l \le N$:

1. for any $1 \le n \le H_l$ there is some $m < l$ such that $A_l^n \subseteq A'_m$;
2. $A'_l \cap A_l^n = \emptyset$ for any $1 \le n \le H_l$; and
3. $A'_N = A$.

The idea behind this is to prove that some parts of the invariant (indexed by A'_l) can be proven true at iteration t, assuming that some other parts of that invariant hold at iteration $t-1$ (those indexed by A_l^0) and also assuming some other parts at t (indexed by A_l^n). There may be several groups of properties to be assumed at iteration t, and the number of these groups is H_l for each l-th subgoal. To avoid cyclic dependences, we have to make sure that we do not assume any component at t that we also want to prove in the same subgoal (condition 2). In addition, every assumed subgoal at t must be proven by an earlier subgoal (condition 1). There are no any restrictions on the components assumed at time $t-1$ (indexed by A_l^0), since these assumptions are part of the inductive hypothesis provided by the induction scheme. And finally, the last condition 3 ensures that we derive the entire invariant at iteration t.

Note, that "iteration" t does not necessarily mean "time" t, and the induction is not necessarily the standard natural induction over time.

This transformation can also be thought of as an application of a bounded induction on A with a measure l with a well-founded order. The base of this induction is $l = 0$ where $\phi^+(\bigwedge_{k\in A'_0}\psi_k)$ has to be derived solely from $\bigwedge_{i\in A_0^0}\psi_i$. Since $\phi^+(X)$ is monotonic in X, the property

$$\phi^+(\bigwedge_{i\in A_1}\psi_i) \to \phi^+(\bigwedge_{i\in A_2}\psi_i) \tag{3.6}$$

holds if $A_2 \subseteq A_1$, and we can use any formulas $\phi^+(\bigwedge_{k\in A'_m} \psi_k)$ for $m < l$ or their corollaries from (3.6) as induction hypotheses in proving $\phi^+(\bigwedge_{k\in A'_l} \psi_k)$. Moreover, if we can show that $\phi^+(\bigwedge_{i\in B} \psi_i) = \bigwedge_{i\in B} \phi^+(\psi_i)$ for any $B \subseteq A$, then we can always have $H_l = 1$, and the condition 1 can be weakened to

1′. $A_l^1 \subseteq \bigcup_{m<l} A'_m$.

Later we refer to the assumptions indexed by A_l^0 as *assumptions with time delay*, and to those indexed by A_l^n as *zero delay assumptions*. A proof rule for this type of circular reasoning is

$$\frac{M;\, \Sigma;\, \Gamma,\, \bigwedge_{i\in A_l^0} \psi_i \wedge \bigwedge_{n=1}^{H_l} \phi^+(\bigwedge_{j\in A_l^n} \psi_j) \,\vdash\, \phi^+(\bigwedge_{k\in A'_l} \psi_k) \quad 0 \le l \le N}{M;\, \Sigma;\, \Gamma,\, \bigwedge_{i\in A} \psi_i \,\vdash\, \Delta,\, \phi^+(\bigwedge_{i\in A} \psi_i)}\mathrm{Circ}$$

and conditions 1, 2, and 3 hold. When A is finite, this rule is trivially derivable from the rest of the system by applying Cut many times. When A is infinite, the conjunction turns into the universal quantifier, and the rule becomes

$$\frac{M;\, \Sigma;\, \Gamma,\, \forall i.\, \psi_i,\, \forall j.\, j < a \to \phi^+(\forall k \le j.\, \psi_k) \,\vdash\, \Delta,\, \phi^+(\forall k \le a.\, \psi_k)}{M;\, \Sigma;\, \Gamma,\, \forall i \in A.\, \psi_i \,\vdash\, \Delta,\, \phi^+(\forall i \in A.\, \psi_i)}\forall\mathrm{Circ}^a$$

for some well-founded partial order $<$ on A, and this rule can be derived from $\mathrm{wf\_Ind}_R^a$. In this case the partial order encodes the dependencies among ψ_i's and, since it is a partial order, ensures that there are no cyclic zero delay dependencies.

"Strong" induction on time. For certain frequently used fixpoints like $\mathbf{G}\,\phi$ one can derive an even stronger rule that utilizes a *strong* induction over time. Observe that the following equality holds:

$$\mathbf{G}\,\phi \equiv \neg\phi\,\mathbf{R}\,\phi,$$

which is quite easy to derive from the fixpoint characterizations of the two LTL operators in the linear time μ-calculus:

$$\mathbf{G}\,\phi \equiv \nu X.\, \phi \wedge \circ X$$

$$f\,\mathbf{R}\,g \equiv \nu X.\, g \wedge (f \vee \circ X).$$

Substituting $\neg\phi$ and ϕ for f and g respectively in the last equation, we get

$$\neg\phi\,\mathbf{R}\,\phi \equiv \nu X.\, \phi \wedge (\neg\phi \vee \circ X),$$

which after propositional simplification yields

$$\nu X.\, \phi \wedge \circ X,$$

precisely the fixpoint for $\mathbf{G}\,\phi$.

The formula $\neg\phi_1\,\mathbf{R}\,\phi_2$ can be intuitively interpreted as the (strong) induction on time where ϕ_1 is the inductive hypothesis and ϕ_2 is the conclusion of the induction step. This is because the semantics of the $\mathbf{R}$ operator dictates that ϕ_2 must hold up until and *including* the moment ϕ_1 becomes false. In other words, if ϕ_1 holds all the time up until $t-1$ (that is, $\neg\phi_1$ has not become true yet), then ϕ_2 must hold up to time t. We stress this further by introducing a new notation, a "temporal implication" operator:

Definition 3.8.1. Define a *temporal implication* operator as follows:

$$\phi_1 \rightarrow_{<t} \phi_2 \equiv \neg\phi_1\,\mathbf{R}\,\phi_2,$$

or, equivalently, in the μ-calculus:

$$\phi_1 \rightarrow_{<t} \phi_2 \equiv \nu X.\,\phi_2 \wedge (\phi_1 \rightarrow \circ X).$$

Intuitively, $\phi_1 \rightarrow_{<t} \phi_2$ means that if ϕ_1 holds up to time $t-1$, then ϕ_2 holds at time t.

Examine now the properties of this new operator. The following lemma states that it allows for *weakening* and is *transitive*, just like the normal implication.

Lemma 3.8.2. *The $\cdot \rightarrow_{<t} \cdot$ operator satisfies the following properties:*

1. *If $\phi_1 \rightarrow_{<t} \phi_2$ in some state s of the model M and $\phi' \rightarrow \phi_1$ is an invariant (that is, $\mathbf{G}\,(\phi' \rightarrow \phi_1)$ holds in the model M), then $\phi' \rightarrow_{<t} \phi_2$ holds in the same state s; and*

2. *If $\phi_1 \rightarrow_{<t} \phi_2$ and $\phi_2 \rightarrow_{<t} \phi_3$ hold in some state s of M, then $\phi_1 \rightarrow_{<t} \phi_3$ holds in the same state s.*

Proof. **Property 1.** We prove this by constructing a derivation in our proof system of the following sequent:

$$M;\,\Sigma,\,\phi' \rightarrow \phi_1;\,\Gamma,\,\phi_1 \rightarrow_{<t} \phi_2 \vdash \Delta,\,\phi' \rightarrow_{<t} \phi_2.$$

1. $M;\,\Sigma,\,\phi' \rightarrow \phi_1;\,\Gamma,\,\phi_1 \rightarrow_{<t} \phi_2 \vdash \Delta,\,\phi' \rightarrow_{<t} \phi_2$ (expand defs, ν_L^α,)
   Expand the definitions of the $\rightarrow_{<t}$ operator into the fixpoints and eliminate them on the left.

2. $M;\,\Sigma,\,\phi' \rightarrow \phi_1,\,\alpha \rightarrow f_1(\alpha);\,\Gamma,\,\alpha \vdash \Delta,\,\nu Y.\,\phi_2 \wedge (\phi' \rightarrow \circ Y)$ (ν_R, 3, 4),
   where $f_1(\alpha) = \phi_2 \wedge (\phi_1 \rightarrow \circ\alpha)$.
   Eliminate the fixpoint on the right; make the "inductive invariant" ψ of the ν_R rule be α. This generates two subgoals: the "base" (3) and the "induction step" (4).

3. $M;\, \Sigma,\, \phi' \to \phi_1,\, \alpha \to f_1(\alpha);\, \Gamma,\, \alpha \vdash \Delta,\, \alpha$ (Init)
   The "base" of the "induction" is trivially true by the Init rule.

4. $M';\, \Sigma,\, \phi' \to \phi_1,\, \alpha \to f_1(\alpha);\, \alpha \vdash \phi_2 \wedge (\phi' \to \circ\alpha)$ (w_I, $\to_L$, expand f_1, $\wedge_L$, 5)
   where $M' = (S,\, \to,\, S)$, the same as M, only the set of initial states is now the entire set of states S.
   In this step we copy the invariant assumptions $\phi' \to \phi_1$ and $\alpha \to f_1(\alpha)$ into Γ and propositionally simplify the sequent. Since α is present in the assumptions by itself, we can derive the conclusion of the second implication $f_1(\alpha)$, which becomes another assumption in Γ. We then expand the definition of $f_1(\alpha)$ and "flatten" its top-level conjunction.

5. $M;\, \Sigma';\, \phi' \to \phi_1,\, \phi_2,\, \phi_1 \to \circ\alpha \vdash \phi_2 \wedge (\phi' \to \circ\alpha)$ ($\wedge_R$, $\to_R$, $\to_L$, Init)
   The resulting sequent is proven by propositional reasoning: ϕ_2 is present on the left, therefore is discharged from the right hand side; the now top-level implication on the right is "flattened," and ϕ' becomes an assumption (the $\to_R$ rule); from ϕ' and the other implications in the assumptions $\circ\alpha$ is derived, which concludes the proof of claim 1.

Property 2. We prove this claim similarly by constructing a derivation of the following sequent in our proof system:

$$M;\, \Sigma;\, \Gamma,\, \phi_1 \to_{<t} \phi_2,\, \phi_2 \to_{<t} \phi_3 \vdash \Delta,\, \phi_1 \to_{<t} \phi_3.$$

1. $M;\, \Sigma;\, \Gamma,\, \phi_1 \to_{<t} \phi_2,\, \phi_2 \to_{<t} \phi_3 \vdash \Delta,\, \phi_1 \to_{<t} \phi_3$ (Expand the $\to_{<t}$ operators, 2)

2. $M;\, \Sigma;\, \Gamma,\, \nu X.\, f_1(X),\, \nu Y.\, f_2(Y) \vdash \Delta,\, \nu Z.\, f_3(Z)$ (ν_L^{α}, ν_L^{β}, 3),
   where $f_1(X) = \phi_2 \wedge (\phi_1 \to \circ X)$, $f_2(Y) = \phi_3 \wedge (\phi_2 \to \circ Y)$, and $f_3(Z) = \phi_3 \wedge (\phi_1 \to \circ Z)$.
   Eliminating the fixpoints on the left, introducing new Skolem constants for predicates α and β (notice that the super-index in the rule names ν_L^{α} and ν_L^{β} is a parameter, it is not part of the rule name *per se*).

3. $M;\, \Sigma,\, \alpha \to f_1(\alpha),\, \beta \to f_2(\beta);\, \Gamma,\, \alpha,\, \beta \vdash \Delta,\, \nu Z.\, f_3(Z)$ (ν_R, 4, 5)
   Now eliminate the fixpoint on the right, taking the formula $\alpha \wedge \beta$ as an inductive invariant ψ in the ν_R rule. This generates two subgoals: the "base" (4) and the "inductive step" (5).

4. $M;\,\Sigma,\,\alpha \rightarrow f_1(\alpha),\,\beta \rightarrow f_2(\beta);\,\Gamma,\,\alpha,\,\beta \vdash \Delta,\,\alpha \wedge \beta$ ($\wedge_R$, Init)
   The "base" of the induction is trivially discharged by propositional reasoning and the Init rule.

5. $M';\,\Sigma,\,\alpha \rightarrow f_1(\alpha),\,\beta \rightarrow f_2(\beta);\,\alpha,\,\beta,\,\alpha \wedge \beta \vdash f_3(\alpha \wedge \beta)$ (w_I, 6),
   where $M' = (S,\,\rightarrow,\,S)$.
   We now copy the two implications from Σ to Γ (the w_I rule) and propositionally simplify the sequent. Namely, since we have α and β as assumptions in Γ, the two implications simplify to just $f_1(\alpha)$ and $f_2(\beta)$ respectively.

6. $M';\,\Sigma';\,\alpha,\,\beta,\,f_1(\alpha),\,f_2(\beta) \vdash f_3(\alpha \wedge \beta)$ (expand f_i's and "flatten" with $\wedge_L$, 7)
   Here $\Sigma' = \Sigma,\,\alpha \rightarrow f_1(\alpha),\,\beta \rightarrow f_2(\beta)$. In this step we expand the definitions of f_i's and eliminate the top-level conjunctions on the left.

7. $M';\,\Sigma';\,\alpha,\,\beta,\,\phi_2,\,\phi_3,\,\phi_2 \rightarrow \circ\alpha,\,\phi_3 \rightarrow \circ\beta \vdash \phi_3 \wedge (\phi_1 \rightarrow \circ(\alpha \wedge \beta))$ (propositional simplification rules: $\rightarrow_L$, $\wedge_R$, $\rightarrow_R$, and Init, 8)
   Propositionally simplify the sequent; this involves both bringing the conclusions of the implications $\circ\alpha$ and $\circ\beta$ on the left directly into the set of assumptions. Additionally, ϕ_3 on the right is discharged and eliminated, since it is also present on the left. This brings the implication on the right to the top level, and it is "flattened" with the $\rightarrow_R$ rule.

8. $M';\,\Sigma';\,\alpha,\,\beta,\,\phi_2,\,\phi_3,\,\circ\alpha,\,\circ\beta,\,\phi_1 \vdash \circ(\alpha \wedge \beta)$ ($\circ\wedge_R$, 9)
   Lift the conjunction from inside the $\Box$ operator.

9. $M';\,\Sigma';\,\alpha,\,\beta,\,\phi_2,\,\phi_3,\,\circ\alpha,\,\circ\beta,\,\phi_1 \vdash \circ\alpha \wedge \circ\beta$ ($\wedge_R$, Init)
   The resulting sequent is proven by simple propositional reasoning.

This completes the proof of the lemma. $\Box$

Lemma 3.8.2 is important since it effectively allows the decomposition of the property $\mathbf{G}\,\phi$ into smaller components in exactly the same way as in the Circ rule. Additionally, the claim 1 in this lemma provides a way to mix the temporal implication with the logical one, thus giving a way to assume some properties at the *current* time t while proving ϕ at the same time t.

The rule for the "strong induction" over time can be formulated in a similarly general way as the rule Circ, and in such a form it will correspond directly to the induction principle introduced in [McM98]. Note, that the formula ϕ in $\mathbf{G}\,\phi$ may contain other temporal operators and fixpoints. In particular, the *liveness property* of the form $\mathbf{G}\,\mathbf{F}\,\phi$ can be proven using the strong induction rule.

These rules may look too complex for the use in any automated theorem prover due to the huge amount of nondeterminism. However, there is a good heuristic for picking assumptions discovered by Ken McMillan [McM98]. In practice, each conjunct ψ_i usually has a form $s_i' = t_i$ or $s_i = t_i$, where s_i is a state variable and t_i is a term, and s' means the value of s in the next state.[4]If the term t_i depends on other state variables s_j for $j \in B \subseteq A$, then we take corresponding ψ_j as our assumptions. If $\psi_i \equiv s_i' = t_i$, then we simply make $A_l^0 = B$ and do not create any zero delay assumptions. In the case when $\psi_i \equiv s_i = t_i$ we try to split B into A_j' for $j < i$, and if we cannot cover B completely, the rest of the indices go into A_i^0. This heuristic seems to work pretty well in practice. A similar algorithm is implemented in the Cadence version of SMV described in [McM98].

3.8.1 Example: Token Ring

Algorithm 2 Token ring.

```
module tokenRing[N: nat] =
begin
  stateVar t: 0..N-1 → bool
  init(t) := λx. if x=0 then true else false endif
  next(t) := λx. if x=0 then t(N-1)
                 else t(x-1) endif

  theorem live = self |= A(G ∀i: F t(i))
end
```

Consider a simple token ring program (Algorithm 2) taken from [McM99]. The original property (or `theorem` in the code) is stated in LTL:

$$\mathbf{G}\,\forall i.\, \mathbf{F}\, t(i).$$

As before, we rewrite the property in the μ-calculus using the translation from Section 2.1.2 on page 31 obtaiting the following theorem.

Theorem 3.8.3.

$$\nu X.\, (\forall i.\, \mu Y.\, t(i) \vee \circ Y) \wedge \circ X.$$

[4] $s' = t$ can be encoded in our logic as $\forall x.\, t = x \to \circ(s = x)$.

Since the specification was in the first order LTL, we have to use linear time μ-calculus and the corresponding proof system for it.

Proof. Since the elements in the array t depend on each other cyclically, we will use our inference rules for cyclic reasoning.

1. $M' = (S, \to, S)$, $\mu Y.\, t(N-1) \vee \circ Y$, $b = 0 \vdash_p \psi_0$ (MC after abstraction $\{0, N-1, \perp\}$)

2. M', $\psi_{b-1} \vdash_p \psi_b$, $b = 0$ (MC after abstraction $\{b-1,\, b,\, \perp\}$)

3. M', $\forall i.\, \mu Y.\, t(i) \vee \circ Y$, $\forall j < b.\, \psi_j$, $b = 0 \vdash_p \psi_b$ (s_L, $\forall_L$, 1)

4. M', $\forall i.\, \mu Y.\, t(i) \vee \circ Y$, $\forall j < b.\, \psi_j \vdash_p \psi_b$, $b = 0$ (s_L, $\forall_L$, 2)

5. M', $\forall i.\, \mu Y.\, t(i) \vee \circ Y$, $\forall j < b.\, \psi_j \vdash_p \psi_b$ (Cut, 3, 4)
   where $\psi_n \equiv \circ(\forall i \leq n.\, \mu Y.\, t(i) \vee \circ Y)$

6. M', $\forall i.\, \mu Y.\, t(i) \vee \circ Y \vdash_p \circ(\forall i.\, \mu Y.\, t(i) \vee \circ Y)$ ($\forall\mathrm{Circ}^a$, 5)

7. M', $\forall i.\, \mu Y.\, t(i) \vee \circ Y \vdash_p \forall i.\, \mu Y.\, t(i) \vee \circ Y$ (Init)

8. M, $\mu Y.\, t(a) \vee \circ Y \vdash_p \mu Y.\, t(a+1) \vee \circ Y$ (COI, MC after abstraction $\{a,\, a+1,\, \perp\}$)

9. M, $\cdot \vdash_p \mu Y.\, t(0) \vee \circ Y$ (COI, MC after abstraction $\{0,\, \perp\}$)

10. M, $\cdot \vdash_p \forall i.\, \mu Y.\, t(i) \vee \circ Y$ (Ind^a_R, 8, 9)

11. $M' = (S, \to, S)$, $\forall i.\, \mu Y.\, t(i) \vee \circ Y \vdash_p \forall i.\, \mu Y.\, t(i) \vee \circ Y \wedge \circ(\forall i.\, \mu Y.\, t(i) \vee \circ Y)$ ($\wedge_R$, 6, 7)

12. M, $\cdot \vdash_p \nu X.\, (\forall i.\, \mu Y.\, t(i) \vee \circ Y) \wedge \circ X$ (ν_R, 10, 11)

$\square$

Even though the structure of the proof may seem similar on surface to the one in [McM99], there is an important distinction from our proof. McMillan uses a *strong* induction on time (assuming the property *at all times up to* $t-1$, prove that it also holds at t), and our proof is by the standard induction which assumes the property at time $t-1$ to prove it at time t. In particular, due to this distinction, the Circ^a rule in step 5 is not necessary, since we can use $\mu Y.\, t(i) \vee \circ Y$ as a hypothesis to prove $\circ(\mu Y.\, t(i+1 \bmod (N-1)) \vee \circ Y)$, and this hypothesis is already present in the sequent after application of ν_R, so the proof gets even shorter. In fact, this is what our heuristic tells us to do in the first place.

In step 2 we do not put value 0 in the abstraction, since it only occurs in the last equality $b = 0$, and this equality can be treated not as a formula but as a statement that $b-1$ is well-defined. In fact, this is the only purpose of splitting the case on $b = 0$. That is, $b = 0$ on the right is equivalent to $b > 0$ on the left in our example, and if we have $b - 1$ in our abstraction (which must be ≥ 0), then this property is enforced by the abstraction. Thus, $b = 0$ will always be false in the abstracted model, and we can safely remove it from the sequent without losing the completeness. In general, however, we might not always be able to carry out this reasoning, and therefore, we'll have to split the cases on whether $b-1 = 0$ and generate two abstractions: $\{0, b-1, b, \perp\}$ and $\{0, 1, \perp\}$.

Chapter 4

Implementation: *SyMP*

SyMP stands for "*Symbolic Model Prover*" and is a general purpose *prover generator* for generating special purpose theorem provers in various application domains. The core of the tool is a generic prover which is connected to several *proof system modules*. Each such module defines an input specification language, a proof system, and a rule application mechanism, and the generic prover provides all the proof management and an interactive user interface.

We will discuss only two proof systems in *SyMP*: the *default* proof system, and *Athena*. The default proof system implements a general framework for combining model checking and theorem proving and has a hardware-oriented specification language that resembles SML. The main purpose of the language is to provide a convenient environment for fast and clean prototyping of new (mostly hardware) verification methodologies based on model checking with some elements of theorem proving. It can also be used as an intermediate representation in translations between other specification languages.

The Athena proof system is specialized to verification of security protocols, and is based on the *Athena* technique developed by Dawn Song [SBP01].

Historically, the default proof system gave the name to the tool. The name "*Model Prover*" is a term we use for a tool that extends theorem proving techniques with the ability to represent and reason about specialized problem domains in their most natural representation. In particular, formal verification normally states its problems as *general model checking* problems; thus, model proving, in particular, provides a systematic way of combining model checking and theorem proving procedures in a single framework. Mathematically, the verification problem remains the same as in model checking: given a model, prove that it satisfies a given property. However, the term "*model checking*" usually assumes that such a proof is done automatically (*checking* tends to mean *checking algorithmically*). Despite several breakthroughs in the past decade, the average size of a model that can be automatically verified is still hardly larger than 10^{100} states, which, though larger than the number of protons in the Universe, still does not go far beyond 300 bits of state variables. This figure is absurdly small compared to the size of a typical modern microprocessor consisting of tens or even hundreds of millions of transistors implementing hundreds of thousands of latches. For those designs the use of elaborate abstractions, and thus, some theorem proving, is inevitable. This, of course, makes the verification task not as automatic, and therefore, cannot be called just "model checking." It is also not a traditional general-purpose theorem proving, since we prove the properties relative to a specific model, and not in general, and heavily use domain-specific techniques. We will refer to this approach as "*model proving*."

The default proof system of *SyMP* is designed to support most of the existing model proving techniques. It has a clean and, hopefully, unambiguous[1] input language also called *SyMP*. It is open for many extensions, so that many new verification methodologies can be implemented in it relatively easily. *SyMP* is best viewed as a front-end to other lower-level techniques, such as explicit state model checking, BDDs, SAT and other decision procedures. The back-end APIs are well-defined and often customizable. In short, it is designed to be "researcher-friendly" in as many ways as possible.

The default *SyMP* language is largely based on ML. There are several reasons for this choice. First, a functional strongly typed language is best suited for theorem proving; and ML has been widely accepted as a convenient implementation language for theorem provers. Since we have to use elements of theorem proving, it is an important consideration. ML does not make the model checking part more difficult, on the contrary, many constructs can be expressed in ML much more elegantly than, say, in *SMV*. Higher-order functions with pattern matching alone can dramatically reduce the size of the program, and at the same time increase its readability. *SyMP* is type-safe, thus, there will be no "run-time" errors due to type mismatches which are annoying and often difficult to debug. In addition, a very popular version of data value abstraction requires the model to be well-typed, and thus, a good type system is a bare necessity for the language. From the implementation point of view, ML is a small yet very expressive language with clean semantics, which makes it easier to implement compared to the HDL languages used in industry, like Verilog or VHDL.[2] Also, guaranteed type correctness allows a more efficient binary encoding of a model into BDDs, which surely helps to increase the efficiency of verification. And the

[1]Unambiguous from the user point of view; that is, the anticipated semantics from the way the program looks must match the reality as closely as possible. The formal semantics must, of course, be well defined.

[2]This, of course, raises an issue of translation from HDLs to SyMP, however, it should be possible to develop an automatic translator for that purpose.

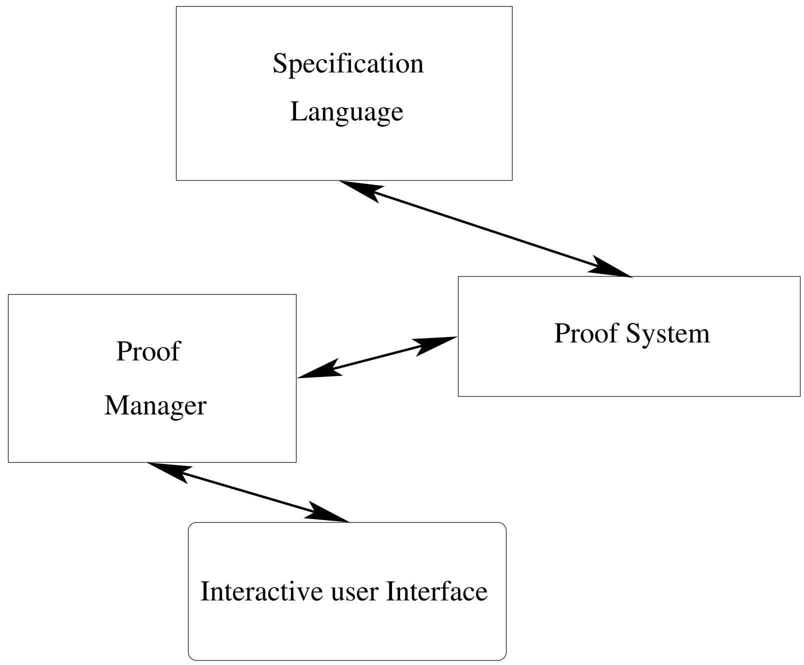

Figure 4.1: The architecture of a typical theorem prover.

last, but not the least reason is that we like ML.

The tool does not have a built-in model checker. Instead, it uses existing external model checkers. In particular, it supports *SMV* [McM93] as a back-end. It is also fairly easy to add support for other model checkers, but *SMV* proved to be sufficient in most cases.

4.1 The Generic Prover

As we have already mentioned, *SyMP* is a *prover generator*. That is, it is a tool that takes an implementation of some basic parts of a proof system and provides a full-fledged interactive theorem prover for that system.

Before we go into the details, let us take a look at a typical architecture of an interactive theorem prover outlined in Figure 4.1. The ultimate task of any theorem prover is to provide a computer-aided support for constructing valid proofs of the user supplied theorems in a particular proof system. These theorems must be somehow communicated to the prover, and this is done through a *specification language*. After the input is processed and converted into some internal representation, the prover must be able to transform it according to the proof system it was designed for. The *proof system module* provides the collection of sound transformations available to the prover.

Theoretically, this is already sufficient to start proving theorems by just applying transformations to the statements (or *formulas*) from the input language until we reach an axiom. However, it is also desirable to remember the sequence of steps we have taken so far, and at the end save it to a file. This sequence of steps constitutes a *proof*, or a *derivation*, and is the final result of the theorem prover that we are looking for.

Additionally, the construction of a proof is often a trial-and-error process, and keeping track of the current partial proof helps to step back and redo certain parts of it as the proof is being constructed. Moreover, the theorem itself may evolve with time, as it is often the case in formal verification. Hence, proof editing and maintenance is another important feature of a practically useful theorem prover. This functionality is provided by the *proof manager module*, which maintains the current proof tree and takes care of editing and saving the proof, as well as checking it for completeness and dependencies on other theorems and definitions.

Finally, the last important component is the *interactive user interface*, the communication channel that allows the user to operate all the features of the theorem prover conveniently.

4.1.1 *SyMP* as a Theorem Prover Generator

In Chapter 3 we have introduced our new proof system for combining model checking and theorem proving . One of the most important reasons this proof system is successful in efficiently combining the two approaches and simplifying correctness proofs of our examples compared to the theorem provers based on classical logics is the narrow specialization of the proof system itself to the problem domain. In our case, the problem domain is parameterized hardware designs and their temporal properties.

Obtaining efficiency through specialization is, in fact, quite a universal approach. In particular, it is natural to expect that other problem domains in formal verification can be dealt with more efficiently if one would design a specialized proof system for them. Indeed, many different specialized proof systems have been developed for solving various verification and theorem proving problems [Koz83, PS99, BBC$^+$99, AJS98, CZ93, ASW94]. Some of them have been implemented as stand-alone theorem provers or automated solvers, others were embedded into existing tools, but many still remain only theoretical achievements on paper.

Although we have seen many new proof systems appear over the past decade, one of the reasons we do not see the same proliferation of new specialized theorem provers is the implementation complexity. This is especially true for the interactive theorem provers. Of

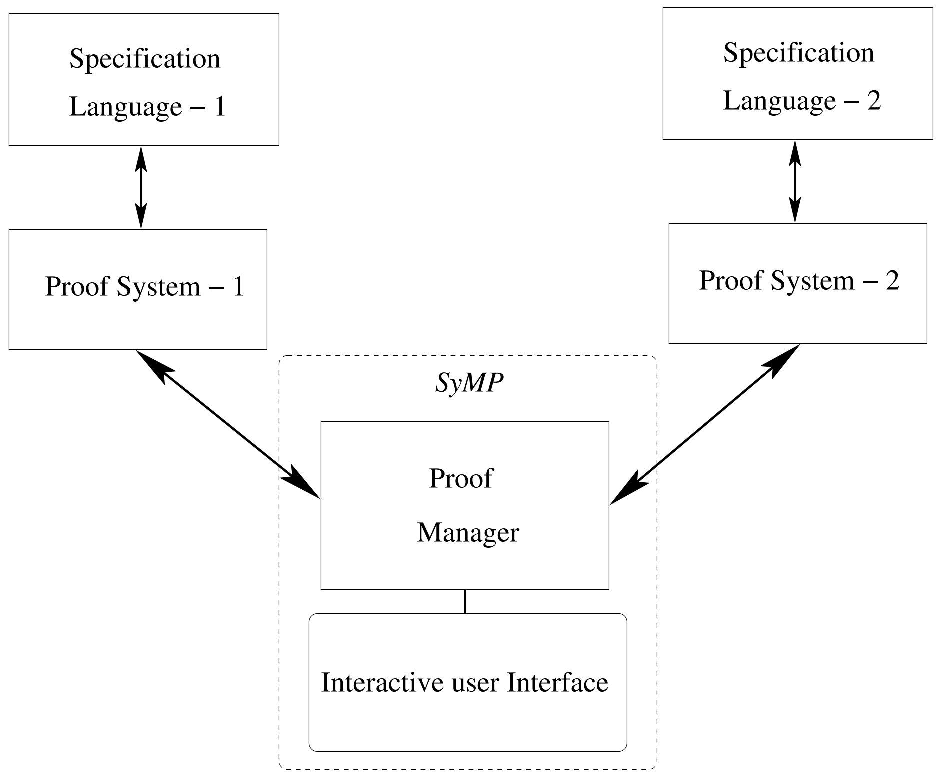

Figure 4.2: The architecture of the *SyMP* prover with multiple proof systems.

course, one can argue that if a completely new proof system needs to be implemented, and embedding it into another existing prover is too inefficient or even practically impossible, then there is little choice but to start a new theorem prover from scratch.

This is not exactly true. A proof as an abstract concept is independent from a proof system, and it is only the proof system itself that is different in those specializations. The interactive user interface is also a completely separate entity which is even out of the scope of the proof theory and only exists in the implementation. Therefore, it is possible to make the proof manager and the user interface modules in Figure 4.1 *independent from the proof system*.

This is precisely the idea behind the architecture of our tool *SyMP* shown on Figure 4.2. The proof manager and the user interface completely independent from any concrete proof system comprise the core of the tool. Proof systems are attached to this core through a well-defined interface (Figure 4.4)

Each proof system module implements a parser and a typechecker for its input specification language, the *sequent* and *inference rule* types, and the rule application function (see Figures 4.3 and 4.4), and the prover provides the rest: the proof tree (actually, the proof

DAG) maintenance and editing, the user interface, and automatic or semi-automatic tactics and proof search engines. The tool can have unlimited number of different proof systems at the same time, however, only one proof system can be active at a time. Even though proof systems reside in one and the same tool, the user is allowed to use only one proof system of his or her choice for each verification example. For instance, it is not possible to use *Athena* proof rules to discharge some subgoals in a proof based on the *default* proof system. The reason for this is soundness: while each proof system can be (and should be!) sound by itself, its combination with other proof systems may become unsound.

The interactive user interface is another module with a well-defined API, and any number of actual interfaces may co-exist. Only one such module can be active at a time due to existing implementation limitations at the time of writing. However, unlike with proof systems, nothing should prevent the user from working in several different environments at the same time on the same proof. Examples of interactive user interfaces are the *emacs-based interface* and the *Java graphical user interface*, both are implemented in *SyMP*, and the latter is contributed by users other than the main developer.

This architecture implies the following from the user point of view. Different proof systems have the same set of user interfaces, the same set of proof management commands, and even the same syntax of the proof rules. The only difference will be the proof rules themselves including the sequent, and the input specification language.

4.1.2 Adding a New Proof System

Suppose some Joe Hacker has created a new proof system for a particular problem domain he is working on. The proof rules are neatly written on a piece of paper, together with a thorough proof of soundness, and Joe even has an idea of an automatic proof strategy which should be able to prove many interesting examples automatically. But he is not completely sure if the strategy or even the proof system itself is actually practical on large real examples, and ideally, an implementation with an interactive proof construction would be really helpful to understand all of its strengths and weaknesses. After some research on the available tools he decided that building a *SyMP* module for his proof system is probably the fastest way to have a working implementation with most of the features he needs. The first question Joe asks us at this point is:

What do I need to do to transfer my proof system from paper to SyMP?

Just as Figure 4.2 suggests, there are two major steps involved: implementing the customized *input language* and the *proof system* itself.

```
type ParseTree          (* "raw" parse tree *)
type ProgramObject      (* typechecked program *)
(* Read stream of characters and
   construct a parse tree *)
val Parse: InputStream -> ParseTree
(* type check the parse tree *)
val typeCheckProgram: ParseTree -> ProgramObject
```

Figure 4.3: Input language interface of *SyMP*: the most important components.

Implementing the input language. Since in an actual implementation of a theorem prover the formulas and sequents must be somehow entered and displayed, and the format is highly dependent on the particular problem domain and the proof system itself, designing an input language is an integral part of the implementation.

SyMP provides an abstract interface specification that the input language module has to implement, and the most important components are outlined in Figure 4.3. Basically, this module has to be able to read a stream of characters (from a file or from the user's input), parse it, and check that the resulting program or expression is well-formed (*type checking*). At the end the result of `ProgramObject` type is the final well-formed object that can be readily interpreted by the custom proof system. For instance, it can be a HOL theorem statement for the Gentzen style proof system; or in the case of the proof system described in Chapter 3, it is a well-typed transition system together with specifications in the first-order μ-calculus.

Implementing the proof system. Just as the proof system on a piece of paper is simply a list of rules, the proof system module implements a *collection of inference rules* together with the appropriate abstract types for representing sequents and rules. It also provides a function that applies a rule to a sequent and returns the list of new sequents that become the new subgoals in the bottom-up proof construction. These basic components are shown in Figure 4.4.

In this module one only needs to implement the very core of the new proof system, the actual transformations that have to be performed in each rule, without having to worry about how and in which order the rules should be applied in a proof. Moreover, since each rule

```
type Sequent       (* Abstract Sequent type *)
type InferenceRule (* Abstract type for infer-
ence rule *)
datatype Result =  (* Result of rule application *)
             (* Success: return new subgoals *)
   Success of Sequent list
               (* Rule doesn't apply *)
  | Failure

(* Function that applies a rule to a sequent *)
val apply: Sequent * InferenceRule -> Result
(* The list of all the rules in the system *)
val allRules: InferenceRule list
```

Figure 4.4: The main part of the interface a proof system module has to implement to connect to the *SyMP* proof manager.

is a separate value, or *object* (even though SML, the implementation language of *SyMP*, is not object-oriented), each rule can be implemented independently of each other. Thus, the implementation has a very high modularity, and further enhancements to the proof system with new rules will not require any changes to the already implemented rules.

Once all the rules are implemented and exported (added to the value of `allRules`), there is only one step remaining to integrate the new proof system into *SyMP*: adding the directory path with the proof system's sources into the central configuration file and recompiling.

Configuring the user interface for the new proof system. In order to interact with the newly created prover in a convenient way, the front-end of the interactive user interface must be configured to recognize the new input language and inform *SyMP* which proof system to use. The concrete details on how to do this depend on the particular front-end used. For instance, with the emacs front-end one has to define a new *major mode* for the input language from the provided templates in emacs-lisp. Other front-ends, especially GUI, may have a simpler graphical configuration utilities for adding a new proof system.

Automated proof search support. Some proof systems are specifically designed to be used with automated proof search, others can be automated to a large extent to spare the user from low-level details of the proof and require only a high-level strategy, and almost none of specialized proof systems are inherently manual with no hope of automation at all.

Automated proof search support is provided in *SyMP* on two levels: in the proof manager (*strategies*) and in the proof system (*tactics*). The difference between the two is rather subtle from the theoretical standpoint, and the only real difference is *how the proof search is specified*, and *where the code for it resides*.

Strategies are general-purpose "meta" rules that are implemented in the central core of *SyMP* and effectively comprise a small programming language to express which concrete inference rules should be tried, in which order, and how failures in rule applications should be handled. For example, the following strategy

```
(try (repeat skolem) flatten split)
```

instructs the prover to try the three steps in that particular sequence until one of them succeeds, at which point the strategy stops, and the proof is updated with the step that has been applied. Notice, that the first step is another nested strategy which will repeat the rule `skolem` as many times as it is applicable to the current sequent, and succeeds if the rule has succeeded at least once.

The advantage of strategies is in their simplicity, generality and flexibility. The strategy mechanism can be readily used with any proof system in *SyMP* to automate the proof search. The "programs" written with strategies are usually very high-level, and therefore, very simple and clean. Additionally, since strategies are written at the interactive user prompt, it is easy to experiment with them while constructing the proof. The last point, however, may become a disadvantage if the strategy becomes rather complicated; not only it can become too inefficient, but also typing it every time may be very inconvenient. In this case, what one really wants is a tactic.

Tactics are implemented and supplied together with the proof system, and therefore, can only be used with that proof system and not with any other. It is implemented similarly to a proof rule, and therefore, has a full access to the internal structure of the sequent. This is an important feature, since a tactic can make its choice of the next inference rule based on some properties of the sequent, while strategies can only have a feedback in the form of success of failure of the inference rules. And finally, a tactic can be as complicated as necessary without any burden on the end user, since it is exported by its name, the same way as an inference rule. Of course, this reduces the flexibility of changing it on the fly and trying different variants in the middle of the proof construction, and this is a part of the trade-off between strategies and tactics.

Implementation complexity. At this point our Joe Hacker may frown and say, "So, basically, I have to implement a new language, all the datatypes and transformations needed for my inference rules, and probably tons of other stuff that is required by the interfaces. But would it really be significantly faster and easier than just implementing a new theorem prover from scratch?"

The answer here is a definite "Yes." This is especially true if the new proof system is quite simple and the rules by themselves are easy to implement. The extra overhead of implementing the required interfaces is quite minimal, and the resulting interactive theorem prover is well worth the efforts. In our experience, after having implemented the input language and the code for the rules, satisfying the interfaces and compiling it into *SyMP* normally takes no more than a few hours.

Even for more involved proof systems like *Athena*, or as sophisticated as the one from Chapter 3, the savings in implementation time and complexity are still quite significant. The reason is that implementing an interactive user interface and a feature-rich proof management module is a complicated and error-prone task. Even though the amount of code for the *SyMP* core is not that great (about 10,000 lines of SML), its complexity and the required level of reliability are rather demanding, which makes it a very valuable shared component to have.

4.2 The Default Proof System[3]

A *SyMP* program typically consists of several, possibly nested, *module declarations* and properties declared as *theorems*. Each module has two types of parameters: *static* and *dynamic*. The *static parameters* allow the user to write a generic description for a whole class of similar devices as one module. Specifying concrete static parameters creates a particular instance of that module. A module can have local theorems in it, and if such a theorem can be proven in general (for any value of the static parameters), then it will also be true for any particular instance of the module. The static parameters can only be instantiated with constant values that can be determined at compile time, or with static parameters of the outer modules. The *dynamic parameters* are used to connect modules with *communication*

[3]Some of the ideas in the *SyMP* language are taken from the *PVS* and *SAL* projects at SRI International. The idea of the proof system itself was mostly inspired by the work of Ken McMillan on his Cadence version of *SMV*.

channels. A channel is simply an expression possibly involving state variables (and, thus, does not have to be computed at compile-time) associated with a formal dynamic parameter of the module. As the name suggests, dynamic parameters can change during an execution, but the module instance does not change with them. What changes are the input values for the same module instance. It is important to make a clear distinction between static and dynamic parameters.

Modules can be composed either *synchronously* or *asynchronously*, and such a composition creates another module. The composition is always written explicitly. In fact, almost every operation and all the dependences among objects in the language are explicit. One of the very few exceptions is the implicit dependency of the state variables on all the static parameters. Read more about it in Subsection 4.2.1.

Expressions in the language closely resemble ML, however, with a few changes. The user can define his own types, constants, and (possibly recursive) functions. Nondeterministic choices are allowed as first class expressions. This means that an arbitrary expression, not only constants, can serve as a nondeterministic choice; and any nondeterministic expression can later be used the same way as any other expression.

SyMP has a *parallel assignment semantics*. Assignments to the state variables are all “executed” in parallel, and can be grouped together under different control structures like **if**, **case**, or **let**. The semantics of the assignments is basically the same as that of *SMV*. Unlike in *SMV*, however, if a variable is not assigned, then its value remains the same in the next step. If you want a variable to change nondeterministically, then an expression `anyvalue` has to be assigned to it. If an error occurs during a computation, a special value `undefined` is returned. The value `undefined` can also be assigned explicitly; this way one can define, for example, partial functions.

It is important to note that the `undefined` value is a special value present in all types, and not a nondeterministic choice of an arbitrary value (like `anyvalue`). For instance, in Java it would correspond to the `null` value. The reason for having this special value is to be able to convert partial functions into total by making them return the `undefined` value whenever the argument is outside of its range. As an example, division is not defined for 0 in the second argument, and therefore, `x/0` evaluates to `undefined`. The alternative is to use predicate subtyping, as is done in *PVS* [SOR93]; there the division operation has type $(\mathcal{N} \times \mathcal{N} - \{0\}) \to \mathcal{N}$, and `x/0` would not be type-correct. However, typechecking then becomes an undecidable problem.

Theorems in *SyMP* are always self-contained, and currently can be of the form $M \models F$ (M is a model of F), where M is a module (which define a *model*, or *Kripke structure*), and

F is a first-order CTL formula. A formula can only depend on state variables defined in the module M on the LHS of $\models$. The proof is not included in the main text of the program and stored separately. A proof can be created automatically or interactively, and the system checks that every rule is applied correctly.

Another important concept is that almost any object in the language has its "literal" definition that can be used interchangeably with the object's name. For instance, a parallel composition of several modules can be declared as another module, and if the same expression for the composition is used elsewhere, it will refer to exactly the same object as the previous named version of it. In particular, two identical instantiations of a module denote exactly the same module (and not the two copies of it!). In order to make distinct copies of a module one needs to change some static parameters in the instantiations. There is more discussion on this design decision later, and in particular, about modules and state variables, since this is the most delicate (and, perhaps, a bit obscure) part.

4.2.1 Language Description

In general, the syntax of the language very closely resembles Standard ML with a few cosmetic changes. Therefore, we assume that the reader is already familiar with the Standard ML and will often describe *SyMP* language features by comparison with SML. Not all of the ML is supported, and there are many additions to the ML core that make *SyMP* a specification language. However, a few fundamental features remain unchanged. The language of expressions is *functional* with *higher-order functions* and *pattern matching*, and *strongly typed* and *type safe* with polymorphic type inference similar to ML.

All identifiers are *case-sensitive*; however, the keywords are not. For example, a variable `X` is syntactically different from a variable `x`, but `statevar`, `stateVar`, and `STATEVAR` are all one and the same keyword.[4]

Comments

SyMP supports both ADA-, or *SMV*-style, and ML-style comments. A comment of the first type starts with '`--`' (double dash) and extends to the end of the line. The ML comment starts with '`(*`' and ends with a matching '`*)`'. The ML-type comments can be nested up to three levels. For instance, the following paragraph is a legal comment in *SyMP*:

```
-- I like SMV comments.
```

[4]This feature is taken after the PVS language. It allows the user to capitalize the keywords as he likes, and at the same time unambiguously reuse the same identifier following some capitalization convention.

```
(* I also like ML comments.
    (* And make them recursive as well.
       (* But only 3 levels deep *) *) *)
```

Including External Files

Some parts of the *SyMP* specification can be kept in separate files and later included in the main file with the '`include`' operator:

```
include("file.symp")
```

This command simply inserts the contents of the specified file, pretty much like `#include` does in C. An included file can also have `include` statements in it; the only requirement is that such an inclusion chain is acyclic.

Types

The built-in types are `bool`, `nat`, and `int`. Notice, that `nat` and `int` are *infinite* types. Finite subranges have the form `[<num_expr>..<num_expr>]`, for example, `[-5..8]`. The unary minus can be either a dash (the same as the binary one) or a tilde like in ML. So, `~5..8` is also allowed. Subranges are subtypes of `int`, and if both bounds are non-negative, also subtypes of `nat`. The type `nat` is a subtype of `int`. There are no user defined subtyping. *Enumerated types*, unlike in ML, do not have to be named. The syntax of an enumerated type is

```
<id> [ of <type_expr> ] { | <id> [ of <type_expr> ] }
```

E.g. a plain enumerated type can be

```
Orange | Apple | IBM
```

A more involved example of a type that introduces a "no value" value to the above type:

```
NONE | SOME of (Orange | Apple | IBM)
```

Here we need to parenthesize the parameter type in order to disambiguate the bars. Without parentheses, the '`of`' would take the precedence over the bar, and the type would have four elements (`NONE`, `SOME`, `Apple`, and `IBM`), and `SOME` will have a parameter of type `Orange`, which most probably doesn't exist in the program.

More complex types can be constructed out the basic types and datatypes. A *record type* has the form

```
'{' <id list>: <type_expr> { , <id list>: <type_expr> } '}'
<id list> ::= <id> { , <id> }
```

For instance,

```
{ n,m: int, mine,yours: MyType,
  other: (one | two | three) }
```

is a record type with five fields.

Tuples have a "product" type of the form

```
<type_expr> { * <type_expr> },
```

and finally, *function types* are constructed by

```
<type_expr> -> <type_expr>.
```

The arrow is right-associative, and one can write

```
int -> bool -> int * bool
```

which is a type of a fully curried function that computes a tuple; it is equivalent to

```
int -> (bool -> (int * bool))
```

The functional type has another form:

```
array <type_expr> of <type_expr>
```

For instance, the last example can also be written as

```
int -> array bool of (int * bool)
```

These two forms define exactly the same type; thus, arrays in *SyMP* are simply functions.

User named types are declared with

```
type <name> = <type_expr>
  -- ';' is optional when end of declaration is unambiguous
type MyType = [-5..5];
datatype 'a list = Cons of 'a * ('a list) | Nil
```

The keywords `type` and `datatype` both define an arbitrary named type; however, `datatype` defines a *recursive type*, so the name of the type being defined can be used inside the type's definition. Unlike in ML, `datatype` is not restricted to enumerated types only, and vice versa, enumerated types (although non-recursive) can be defined with `type` keyword. In the `type` clause the type's name refers to previous definition of that name, thus, no recursion is allowed. Named types may also have *parameters*, exactly as in ML.

After this declaration the name becomes an abbreviation for the type. However, one can still use the original type expression to denote exactly the same type.

Expressions.

Expressions are very similar to ones in ML, and we describe them only briefly. There are, however, a few syntactic changes to some of them that we have to mention.

Scalar Expressions. Scalar expressions are those of types `bool`, `int` and `nat`. Literals are `true` and `false` for the boolean type, and numerals for `int` and `nat`. Boolean operators are connectives `not`, `and` (or `&`), `or`, `implies`, (also -> and =>), and `iff` (or <->). There are no ML-like connectives `orelse` and `andalso`. In addition, *SyMP* has quantifiers with the obvious semantics:

```
forall <var list>: <expr>
exists <var list>: <expr>
<var list> ::= <bound vars> { , <bound vars> }
<bound vars> ::= <id> | ( <id list> : <type_expr> )
```

An example of a quantifier expression is:

```
forall i,j,(p,q: one | two | three), (b: bool):
    F(i+j,b or p = q)
```

Values of any types, including functions, can be compared with each other with =, `!=` and <> operators (the last two are "not equal", and are semantically identical). Arithmetic operators are +, −, `*`, / (divide), `div`, and `mod`. Numerical expressions can be compared with <, >, <= and `>=`.

Datatypes. The values of datatypes are constructed the same way as in ML.

`Anyvalue` and `undefined`. `Anyvalue` stands for the completely nondeterministic choice for the type of its current context. It is an abbreviation for a corresponding nondeterministic expression that simply lists all possible values. However, it can also be used with infinite types, for which no such expression can be constructed otherwise. `Undefined` is a special value that every type has, and it indicates a run-time error, for instance, a division by 0. This value can also be assigned explicitly; this allows the user to define partial functions.

Tuples, Records and Patterns. The value constructors for these types are the same as in ML. We only give a short example involving all of them:

```
let val (first,_) = pair
    val { n = n, ... } = a_record
in (first,n,first=n+3) end
```

Besides the pattern matching, a record field can be extracted, with a '`.`' (dot) operator. For instance, the second local declaration in the above example can be done with

```
    val n = a_record.n
```

Nondeterministic choice. A list of expressions separated by a vertical bar '`|`' means a nondeterministic choice among the values of these expressions. This is a new construct that does not exist in ML. A value of such an expression is a value of one of the expressions chosen nondeterministically.

A nondeterministic expression is a first-class object, that is, it can be used anywhere as any other expression where a nondeterministic choice makes sense (e.g. in computing the values of state variables), and these choices can be nested arbitrarily. The semantics of a nondeterministic choice is a set of values. Any scalar operation like a function application computes the image of the set. A tuple or a record of nondeterministic expressions is a Cartesian product of the corresponding sets. A function that has a nondeterministic definition becomes a relation. Such functions, however, cannot be used to compute any static values.

Functions. The λ-form of a function definition is the same as in ML:

```
fn <pattern> => <expr> { | <pattern> => <expr> }
```

If the match is non-exhaustive, the function becomes partial, and will return `undefined` for the unmatched values.

`val` and `fun` declarations. The declarations `val` and `fun` have the usual ML syntax:

```
<val decl> ::= val <pattern> = <expr>
<fun decl> ::= fun <id> <pattern list> = <expr>
               {| <id> <pattern list> = <expr> }
<pattern list> ::= <pattern> { <pattern> }
```

As with λ-forms, all matches must be exhaustive, otherwise an `undefined` value may be generated. These constructs define named constants. A function declared with `fun` can be recursive. However, mutually recursive functions are not as easy to write because there is no `and` operator as in ML. It is probably a bug in the design, but we decided to use `and` as a boolean operator instead.

Note that when using a polymorphic typing for a parameter in a declaration, such as:

```
fun f (x : 'a) = ... (z : 'a) ...
```

this only ensures that x and z share the same type, rather than (as in Standard ML) forcing x to be a polymorphic type.

`if`, `case`, `let`, and `with` clauses. The first two constructs are slightly different from their ML counterparts. Their formal syntax is:

```
<if expr> ::= if <expr> then <expr>
              { elsif <expr> then <expr> }
              else <expr> endif
<case expr> ::= case <expr> of
                    <pattern> => <expr>
                { | <pattern> => <expr> }
                endcase
<let expr> ::= let <local decl> { <local decl> }
               in <expr> end
<local decl> ::= <val decl> | <fun decl>
```

Notice the closing keyword at the end of each clause. This change is not fundamentally necessary, but it makes the language more structured, and unifies these clauses with the imperative ones for the assignment part, where such delimiters are necessary for grouping the assignments. As in functions, if not all the cases are covered in pattern matching, the appropriate values become `undefined`.

The `with` operator is an extension to ML that is particularly useful. Its syntax is

```
<expr> with '[' <id> := <expr> { , <id> := <expr> } ']'
```

The expression must be of a record type, and the identifiers are the names of the fields. The value of the whole expression is the same record value with the mentioned fields updated with the values of the corresponding expressions.

Modules.

Modules are the most important part of *SyMP*, and their structure is among the key features in achieving a natural composition of many modern verification techniques within one framework.

Module Declaration. A *module declaration* has the form

```
module <id> [ '['<static parms>']' ] [ <pattern> ]
  = <module expr>
<module expr> ::= <begin-end clause>
                | <module instance>
                | <parallel composition>
```

The *header* of the module declaration starts with the keyword `module` followed by the module's name, followed by optional *static* and *dynamic parameters*. The static parameters are enclosed in square brackets to be distinguished from the dynamic ones. Either or both kinds of parameters can be omitted. The static parameters is a comma-separated list of individual formal parameter declarations. Each parameter can be a type or a constant. Type parameters are introduced with the `type` keyword:

```
<static type> ::= type <id>
```

A constant parameter is simply a (possibly typed) identifier:

```
<static const> ::= <id> | <id> : <type_expr>
```

If a static constant parameter is untyped, its type will be inferred from the context, as in any other declaration. The type expression for a constant may include types declared earlier in the list of parameters. Constants can also be grouped together if they have the same type.

Dynamic parameters are essentially a single *pattern*. Most often one only needs a simple form of a tuple pattern: a comma-separated list of identifiers. As an example consider the following module header taken from a preliminary implementation of the reference file in Tomasulo's algorithm:

```
module reffile[i: nat, -- index for multiple copies
               type Regs,
               type Inst,
               type T,
               type Tag,
               reg_init: array Regs of T]
              (op: Inst, src1,src2,dest: Regs) = ...
```

Notice, that a constant parameter `reg_init` uses `Regs` and `T` in its declaration, which are type parameters declared earlier. The dynamic parameters can also refer to static parameters; however, dynamic parameters cannot refer to each other. For example, a static parameter of type `nat` can be used to declare a subrange both in the rest of the static block, and in the dynamic one. But a dynamic parameter of the same type cannot be used to define a subrange.

Static parameters are used as constant objects known at compile time; when static parameters of a module are instantiated, a new syntactic copy of the module is generated with concrete values instead of their names. Since it is a syntactic substitution, we can allow static parameters to be as rich as they are, and depend on each other in a non-trivial way. Dynamic parameters, on the other hand, serve the purpose of connecting different modules with each other through *data channels*, and thus, cannot be computed at compile time. Different instantiations of dynamic parameters do not create different module instances; they are merely different inputs to the same module.

Parameterized modules are often used to "encapsulate" theorems about certain parts of a system that can be proven in general for arbitrary values of their parameters. An instantiation of such a module creates particular instances of its theorems that will automatically be true, if proven in general. This provides a way of abstracting irrelevant parts of a component by putting them into the parameters, proving correctness of the simplified version, and then instantiate (or "refine") it to the original, presumably, much more complex configuration, which, thus, will also be correct.

Module Instances. A module instance is the module's name with actual static (in square brackets) and dynamic parameters. For example, the module `reffile` declared above can be instantiated as

```
reffile[3,[0..15],(Plus | Minus),
        [-65536..65535],TagType, Init]
       (Instructions)
```

Here `Instructions` is a variable of type

```
(Plus | Minus) * [0..15] * [0..15] * [0..15]
```

which is a tuple type. This creates a module object implementing a concrete reference file with index 3 (whatever it is), with 16 registers, two operations (`Plus` and `Minus`), 16-bit signed data values, previously defined type of tags and the initial value vector. The module's dynamic input is connected to the variable `Instructions`. Notice, that all the dynamic parameters can be instantiated with one single variable; this is because a module can have only *one* dynamic parameter, but, perhaps, of a complex type. In our case this is a tuple, but in general it can be anything else. All of the parameters specified in the module's declaration are required.

Parallel Composition. There are two types of parallel composition in the current version: *synchronous* and *asynchronous*. Synchronous composition of two modules is given by the infix double bar expression:

```
<module expr> || <module expr>
```

Asynchronous composition is similar, but uses a single bar:

```
<module expr> | <module expr>
```

Both operators are left-associative and have the same priority. Thus, the following expression

```
A || B || C | D || E
```

is the same as

```
(((A || B) || C) | D) || E
```

Modules can also be composed using `sync` and `async` keywords:

```
sync <var list> : <module expr>
async <var list> : <module expr>
```

For instance,

```
sync(i: [1..20]): boo[i]
```

will compose synchronously 20 copies of a module `boo`. The variable `i` is a *bound variable* within the scope of `sync`, and is considered as a constant known at compile time, so that it can be used to instantiate static parameters. One can bind several variables with one such "quantifier", and use any module expression, including nested `sync` and `async`.

```
async(i: [0..9]),(j: nat):
   moo[i] | sync(k: one | two | three): m2[j,k]
```

It is possible to declare a composition of an infinite number of modules.

Only fully defined and fully instantiated modules can be composed together. It is not legal to refer to an outer module from a submodule in a composition, unless the outer module is also one of the modules the composition. The current module, however, can refer to itself if it is defined using `begin-end` clause, and the composition is done inside this clause. In this case, the module is referred to with the keyword `self`, and can be thought of as an instantiation of itself with the module's formal parameters:

```
module M[type T](i: T) =
begin
  ...
  module subM = self || boo[5]
end
```

Begin-End Clause. A `begin-end` clause is a sequence of declarations between matching `begin` and `end` keywords. All kinds of declarations are allowed between `begin` and `end`, including module declarations, plus declarations and assignments to the *state variables*. State variables are the only objects that can change dynamically with time, and they are the only means to introduce execution in the model; everything else is computed statically at compile time. A declaration of a state variable is of the form

```
StateVar <id> { , <id> } [ : <type_expr> ]
```

There are three types of basic assignments: "*normal*" (or *immediate*, or *invariant*), *initial*, and *next*. Every state variable has to have either a normal assignment, or an initial and the next assignment. A variable can only be assigned once in the program. The last rule has a broad sense; for instance, a variable can syntactically have two "next" assignments, but only if they are located in mutually exclusive execution branches, like one in the "then", and the other in the "else" part of an "if" statement. The syntax of these three assignments is the following:

```
<var_expr> := <expr>
init(<var_expr>) := <expr>
next(<var_expr>) := <expr>
```

A <var_expr> is an expression that refers to a single state variable or its component. We will come back to its more precise definition later. An <expr> is an arbitrary expression, possibly involving state variables. Only state variables of the current module can be assigned; all the other visible state variables can only appear on the RHS of any assignment, including state variables of locally defined modules. Also, the `next` keyword cannot appear on the RHS. This is different, for example, from the CMU version of *SMV*[5] model checker [McM93]. If a state variable has neither normal nor next assignment, then the assignment

```
next(the_var) := the_var
```

is assumed by default, and thus, an initial assignment is required.

A vector of values of all the state variables constitute a *state* of the module. The state of a module does not include the states of its submodules; in order for it to be the case, the

[5]Originally, in *SMV* this was probably an inadvertent feature (read *a bug*) which turned out to be very convenient for formal method hackers who know what they are doing, but dangerous for soundness otherwise. This feature was later made available in a sound form in the Cadence version of *SMV*.

```
next(state) :=
  let val tmp =
       if state = shared then (shared | invalid)
       else state endif
  in ...
    case gbus.trans of
      ...
        invalidate => if gbus.cancel then tmp else ...
      | _ => tmp
    endcase
  end
```

Figure 4.5: An example of the `let` clause from the Futurebus+ specification [IEE94].

current module has to be composed with its submodules, and then this parallel composition will include in its state the states of its components. An *execution* of a module is an infinite sequence of states, such that the first state is an initial state, as defined by the "init" and "normal" assignments, and each subsequent state is related to the previous one by the module's *transition relation* defined by the "normal" and "next" assignments. This is very similar to the semantics of *SMV*.

As it was already mentioned above, *SyMP* has a *parallel assignment semantics*. That is, an execution of a module is performed in *cycles*, and each clock cycle all the variables are updated *simultaneously*. Normal assignments will always keep the value of the assigned variable equal to the RHS of the assignment; in this sense, normal assignments are *invariants* of the transition relation. One can also think of this assignment to take effect immediately, without a time delay. The effect of the "next" assignment is delayed for one clock cycle. That is, the next state in the execution sequence must have the value of that variable equal to the expression on the RHS evaluated in the current state.

Where Are the "DEFINE" Macros[6]? There aren't any. Instead, one can use a `let` clause to give a name to a repeatedly used expression, as shown in Figure 4.5. Also, a similar *imperative* `let` can be used, if the shared expression is used to assign several state variables:

```
let <local decl> { <local decl> }
in <asst list>
```

[6]This is a highly cherished feature of *SMV* to be able to define shared subexpressions.

```
end
<asst list> ::= <asst> { [ ; ] <asst> }
```

Notice, that here the `end` keyword is necessary to indicate the scope of the imperative `let`.

It is also possible to declare a new variable and assign it a repeatedly used value with a "normal" assignment. If the variable deterministically depends on other variables, *SyMP* may replace it internally by a macro, effectively eliminating it from the module's state.

It is important to note that a nondeterministic normal assignment is different semantically from the local named expression. When an expression is given a name in `let`, the name stands for the expression *syntactically*, and is effectively a *macro*. This means that in different instances it may make different nondeterministic choices. For example,

```
let val x = one | two | three
in x = x end
```

does not have to be true, because it is equivalent to

```
(one | two | three) = (one | two | three),
```

which evaluates to `(true | false)` — a boolean nondeterministic choice.

Functions defined with nondeterministic expressions are treated similarly. A function application must be equivalent to its β-reduced form, thus, if a function is defined as

```
fun ff(x,y: nat) = (x | y)
```

then `ff(3,4) = ff(3,4)` is the same as `(3 | 4) = (3 | 4)`, and this, again, does not have to be true.

On the contrary, a state variable must always be equal to itself. Therefore, if one needs the same nondeterministic choice to be used in different places, one has to use a state variable for that. For instance, the first example may become

```
statevar x: one | two | three
x := anyvalue
  ... x = x -- is always true now!
```

```
let val tmp =
     if state = shared then (shared | invalid)
     else state endif
in ...
  case gbus.trans of
    ...
      invalidate =>
       if gbus.cancel then next(state) := tmp else ...
    | _ => next(state) := tmp
  endcase
end
```

Figure 4.6: The same example from the Futurebus+ specification as in Figure 4.5, but with the imperative `let`.

`let`, `if`, `case`, `choose` and `nop`. In general, assignments can be enclosed into `let`, `if`, and `case` statements that have the same syntax as their expression counterparts. The difference, however, is that they are now imperative control structures that return no value, and can only have a sequence of assignments in place of the result expression. For instance, Figure 4.6 shows the same example as in Figure 4.5, but with imperative constructs. If a certain branch should not contain any assignments, then it must contain a keyword `nop`.

Although nondeterminism can be encoded directly into expressions, it is sometimes useful to have another type of it: *guarded nondeterminism*. The `choose` structure is used for that, and its syntax is the following:

```
choose [ <bound vars> : ]
      <expr> => <asst list>
  { | <expr> => <asst list> }
endchoose
```

It has the semantics of *guarded commands*. The `<bound vars>` are optional *nondeterministic parameters*, and the expressions are boolean guards which are all evaluated in parallel. The statement picks nondeterministically some values for the parameters that make at least one guard true, then one of the branches with a true guard is chosen nondeterministically for execution. If all of the guards are false for any values of the parameters, then `nop` is executed. Note, that unlike in `case` and `if` statements, all of the branches are symmetric and can be reordered without changing the meaning of the program. In particular, if there is a true guard at the end, it may still be executed even if there are true guards before it.

Theorems. Theorems are introduced by one of the keywords: `theorem`, `lemma`, `proposition`, `corollary`, `conjecture`, `specification`, or `spec`, all of which are treated the same. The syntax is

```
theorem <module expr> |= <frm>
```

A theorem states the validity of a first-order CTL formula (however, the syntax allows first-order μ-calculus with CTL operators) in a particular module.

A CTL formula is a boolean expression with the usual CTL operators:

```
<frm> ::= <expr> | ex <frm> | ax <frm>
            | eg <frm> | ag <frm>
            | ef <frm> | af <frm>
            | sometimes '[' <frm> until <frm> ']'
            | always '[' <frm> until <frm> ']'
            | sometimes '[' <frm> releases <frm> ']'
            | always '[' <frm> releases <frm> ']'
            | mu <bound var> : <frm>
            | nu <bound var> : <frm>
```

Fairness constraints are not included in the language yet.

Every theorem must be *closed*, that is, the entire cone of influence of a CTL formula must be included in the module on the LHS. Likewise, any module instance in a module expression can only depend on the state variables of those modules that are composed with it.

Visibility Rules: the `export` clause. The visibility rules in *SyMP* resemble those of ML, or Pascal. Any object declared before on the same level or higher in the hierarchy is visible, except for the state variables. By default, all the state variables are visible to the higher-level modules.

A locally defined object can be referred to outside of the module using the dot notation:

```
MyModule[i].x
```

A module can also be "opened" entirely, making all of its objects visible at the current level:

```
open MyModule[i]
```

This declaration makes all the visible identifiers of the module `MyModule[i]` appear as if they were declared at the current level. In other words, an identifier `x` will be treated the same as `MyModule[i].x`. The `open` directive may shadow some identifiers at the current level declared before.

A Few Notes on the Semantics.

The *SyMP* language has a general rule that if two objects are syntactically equivalent, they must be one and the same object semantically. For example, two instances of a module with the same set of static parameters refer to one and the same module instance. Thus, in the code

```
if MyModule[5].x = 5 then b := MyModule[5].x else ...
```

`b` will always be assigned 5 when the condition is true. This rule extends to the named objects as well; for example, the code above is equivalent to

```
module tmp = MyModule[5]
if MyModule[5].x = 5 then b := tmp.x else ...
```

And it is also the same as the following:

```
module tmp[i: nat] =
  begin ...
    module M = MyModule[5]
    export M
  end
if tmp[5].M.x = 5 then b := tmp[8].M.x else ...
```

In the last example the module `M` remains the same because it does not depend on the static parameter `i` of `tmp`; at the same time `tmp[5]` and `tmp[8]` are two *distinct* instances of the module `tmp`, they just happened to share the same submodule. In general, whether two objects are the same or not is judged by going to the core definitions of them and checking whether they are identical or not.

The only "exception" to this rule are state variables; they are assumed to depend on all of the static parameters of the module they belong to. Thus, if `MyModule` is defined as

```
module MyModule[i: nat] =
 begin
   stateVar x: int
   ...
 end
```

then `MyModule[5].x` and `MyModule[20].x` are two distinct state variables living in two different instances of `MyModule`. In other words, state variables cannot be shared across the modules or module instances.

If one needs to have several copies of exactly the same module, one has to introduce a *dummy parameter* to index the instances. Although no object inside the module may explicitly depend on it, the state variables will depend on it implicitly, and thus, will be distinct in different instances.

4.2.2 The Proof Rules

The default proof system is based on the proof system described in Chapter 3, but the rules often take very different forms that makes them more practical or more powerful. This proof system uses the same sequent as in Chapter 3:

$$M;\ \Sigma;\ \Gamma \vdash \Delta,$$

where $M = (S, \rightarrow, I)$ is the model, or a *Kripke structure* given by its set of states S, transition relation $\rightarrow \subseteq S \times S$, and the set of initial states I; and Σ, Γ, and Δ are sets of first-order μ-calculus or CTL formulas. Formulas in Σ are *invariant constraints* on the model, that is, the set of states in the model is restricted to only those satisfying all the formulas in Σ. Such a constrained model is denoted $M|_\Sigma$. Formulas Γ are assumptions that are assumed to hold in the initial state, and at least one formula in Δ must hold in each of the initial states. In other words, a sequent $M;\ \Sigma;\ \Gamma \vdash \Delta$ *holds* iff

$$M|_\Sigma \models \bigwedge \Gamma \rightarrow \bigvee \Delta.$$

We also refer to Σ and Γ as *left hand side* (LHS) of the sequent, and to Δ as *right hand side* (RHS).

Below are some examples of the rules implemented in the default proof system.

`init` Checks if the left hand side of the sequent has any formulas in common with the right hand side, or if there are `false` or `true` formulas on the LHS or RHS respectively, and completes the proof for the sequent. It corresponds to the Init and Init_I

rules, whichever applies to the sequent:

$$\frac{}{M;\, \Sigma;\, \Gamma,\, \phi \vdash \Delta,\, \phi}\mathrm{Init} \qquad \frac{}{M;\, \Sigma,\, \phi;\, \Gamma \vdash \Delta,\, \phi}\mathrm{Init}_I.$$

This rule is applied automatically after any new sequent is generated, and the user normally does not have to use it at all. It is only necessary in the extreme case when a theorem contains the only statement `true`, which is a pretty useless theorem anyway.

cone Perform the *cone of influence* reduction on the model:

$$\frac{M';\, \Sigma;\, \Gamma \vdash \Delta}{M;\, \Sigma;\, \Gamma \vdash \Delta}\mathrm{COI}.$$

This rule is run internally by the **modelcheck** rule.

`modelcheck` If the model in the current sequent is finite and small enough, run an external model checker on it:

$$\frac{}{M;\, \Sigma;\, \Gamma \vdash \Delta}\mathrm{MC} \quad \text{provided that} \quad M'|_\Sigma \models \bigwedge \Gamma \to \bigvee \Delta,$$

where M' is the result of applying the COI rule. So, in effect, this is a compound rule derived from two rules MC and COI.

`flatten` "Flatten" the sequent; that is, apply conjunction on the left, disjunction and implication on the right, and negation rules as much as possible. It is based on the propositional rules from the original proof system that have only one premiss; that is: $\wedge_L$, $\wedge_I$, $\vee_R$, $\to_R$, $\neg_R$, and $\neg_L$. It is called "flatten" because all the formulas in the sequent are split into the smallest possible propositional subformulas without generating more subgoals or increasing the total size of the sequent. The resulting sequent, therefore, becomes easier to work with, as more "interesting" operators (like quantifiers) appear on the top level. These propositional rules are invertible, and since the number of subgoals remains the same, the final subgoal can only become simpler to prove.

split Split conjunction on the right, and disjunction and implication on the left into several subgoals. This rule is a combination of $\wedge_R$, $\vee_L$, and $\to_L$, and together with **flatten**, these two rules cover all the propositional proof rules from the original proof system.

`copy` Copies the specified formula, so it will appear twice in the sequent. This may be necessary if one wants to use the same formula more than once in different ways. It implements the original rules Copy_R, Copy_L, and Copy_I.

`delete` Deletes the specified list of formulas from the sequent (original rules w_R and w_L). This rule helps to keep the proof cleaner, if the deleted formulas are not needed to prove the sequent. Also, if an automatic proof search is used, it may find the proof faster, since it does not have to bother with the irrelevant formulas.

useinvar Copy an invariant formula from Σ to the set of initial state assumptions Γ (original rule w_I).

replace Treats a formula on the LHS of the form $A = B$ as a *rewrite rule* and substitutes B for A everywhere in the sequent where it is sound to do. This rule does not have a counterpart in the original proof system, since it relies on the interpreted equality symbol, which is not present there. It is a very useful extension, since the input language includes the equality operator.

case Split cases on the validity of a given formula. This is the most basic form of the *cut* rule.

forallcase An infinite version of **case**: "splits cases" on all the values of a given term by rewriting a formula A as $\forall x.\, t = x \to A$. It is exactly the implementation of the original $\forall\mathrm{Case}_R$ rule:

$$\frac{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \forall x.\, t = x \to \phi}{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \phi}\forall\mathrm{Case}_R.$$

split_label Split cases on all the possible choices in the given labeled assignment. This rule is sound only for formulas that refer to states at most one clock cycle in the future. It is an extension to the original proof system, and exploits the semantics of the imperative structures in the input language. For instance, if a model contains an assignment of the form:

```
choose
    g1 => A1;
   ...
  | gn => An;
endchoose
```

then an instance of this rule can be written as follows:

$$\frac{M_1;\, \Sigma;\, \Gamma,\, g_1 \vdash \Delta \quad \cdots \quad M_n;\, \Sigma;\, \Gamma,\, g_n \vdash \Delta \quad M_{n+1};\, \Sigma;\, \Gamma,\, \neg g_1,\, \ldots,\, \neg g_n \vdash \Delta}{M;\, \Sigma;\, \Gamma \vdash \Delta}\text{split label},$$

where the only temporal operator in all of the formulas is $\mathbf{AX}$, no nesting of $\mathbf{AX}$ is allowed, and each M_i is obtained from M by replacing the entire `choose` assignment with A_i, except for the last M_{n+1}, where the `choose` assignment is replaced by `nop`. Intuitively, given the restrictions on the formulas, we are only interested in the current and the next states (one transition). The `choose` assignment prescribes that there are $n+1$ ways the model can transition to the next state, depending on which guards are true, or if any of them is true at all. The rule exploits this construct to simplify the model and generate several simpler subgoals.

skolem Eliminates a universal quantifier on the RHS or an existential quantifier on the LHS by instantiating bound variables with new *Skolem constants*. It implements the original $\forall^a_R$ and $\exists^a_L$ rules:

$$\frac{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \phi(a)}{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \forall x.\, \phi(x)}\forall^a_R \qquad \frac{M;\, \Sigma;\, \Gamma,\, \phi(a) \vdash \Delta}{M;\, \Sigma;\, \Gamma,\, \exists x.\, \phi(x) \vdash \Delta}\exists^a_L.$$

inst Eliminates an existential quantifier on the RHS or a universal quantifier on the LHS by instantiating bound variables with the given terms. Implements the original $\exists_R$ and $\forall_L$ rules:

$$\frac{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \phi(t)}{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \exists x.\, \phi(x)}\exists_R \qquad \frac{M;\, \Sigma;\, \Gamma,\, \phi(t) \vdash \Delta}{M;\, \Sigma;\, \Gamma,\, \forall x.\, \phi(x) \vdash \Delta}\forall_L.$$

induct_ag Applies induction over time to an $\mathbf{AG}$-formula on the RHS. This is a derived rule from ν_R, where the fixpoint formula $\nu X.\, \phi(X)$ is replaced with the more concrete form $\nu X.\, p \wedge \Box X$, which is equivalent to $\mathbf{AG}\, p$. Instantiating this formula into ν_R gives:

$$\frac{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \psi \quad M' = (S, \rightarrow\ S);\, \Sigma;\, \psi \vdash p \wedge \Box\psi}{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \nu X.\, p \wedge \Box X}\nu_R,$$

and the second subgoal can be split into two by the $\wedge_R$ rule:

$$\frac{M';\, \Sigma;\, \psi \vdash p \quad M';\, \Sigma;\, \psi \vdash \Box\psi}{M';\, \Sigma;\, \psi \vdash p \wedge \Box\psi}\wedge_R,$$

yielding the new derived rule in its final form, this time all subgoals written directly in CTL:

$$\frac{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \psi \quad M';\, \Sigma;\, \psi \vdash p \quad M';\, \Sigma;\, \psi \vdash \Box\psi}{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \mathbf{AG}\, p}\text{induct\_ag}.$$

The new formula ψ is an *inductive invariant* provided by the user. Note, that this is a "natural" induction on time, when ψ is assumed to hold at time $t-1$, and is proven to hold at time t.

induct_ag_ar Applies "strong" induction over time to an $\mathbf{AG}$ -formula on the RHS. This rule can be derived from the original fixpoint and propositional rules based on the μ-calculus characterization of the $\mathbf{AG}$ and the release CTL operators:

$$\frac{M;\,\Sigma;\,\Gamma\,\vdash\,\Delta,\,\mathbf{A}[\neg\phi\,\mathbf{R}\,\phi]}{M;\,\Sigma;\,\Gamma\,\vdash\,\Delta,\,\mathbf{AG}\,\phi}\mathrm{induct\_ag\_ar}.$$

The new formula $\mathbf{A}[\neg\phi\,\mathbf{R}\,\phi]$ is semantically identical to $\mathbf{AG}\,\phi$, and it may seem that no gain is obtained by rewriting one to the other as it is. However, this rewriting separates the inductive hypothesis (the first $\neg\phi$ component of $\mathbf{AR}$) and the goal of the induction ϕ, so they can be worked on separately. See Section 5.1.2 for practical applications of this rule.

lift_forall_ax Pulls a universal quantifier out of the scope of $\mathbf{AX}$ operator; that is:

$$\frac{M;\,\Sigma;\,\Gamma\,\vdash\,\Delta,\,\forall x.\,\mathbf{AX}\,\phi}{M;\,\Sigma;\,\Gamma\,\vdash\,\Delta,\,\mathbf{AX}\,\forall x.\,\phi}\mathrm{lift\_}\forall\mathrm{\_}\mathbf{AX}\,.$$

abstract_split Applies abstraction to a sequent with Skolem constants to transform it to a finite-state problem before it can be model checked. This rule is quite complex, and we postpone its discussion until we consider the example of the IBM cache coherence protocol in Chapter 5, Section 5.1.

4.3 The Athena Proof System

4.3.1 Strand Space Representation

The *Athena* proof system is designed for the verification of security protocols, and is based on the original work by Dawn Song [SBP01]. It uses *Strand Spaces* [THG98b, FHG98, THG99] as the basis for the protocol representation, which are defined only rather informally in this document, and the reader is referred to [SBP01] for exact details.

Messages.

Security protocols are based on message passing among the participants, or *principals*. In this formalism, messages are *terms* built from *atomic messages* using *concatenation*

$(m_1, m_2, \ldots)$ and *encryption* ($\{m\}_K$) operators (decryption is considered a particular form of the encryption operator). Messages can be of several types: *key* (symmetric, public, or private key) is a message that is used to encrypt or decrypt other messages, *nonce* is a random message meant to be freshly generated in every protocol run to prevent replay attacks, *name* identifies principals, and finally, *message* is a supertype of all of the above, including concatenated and encrypted messages of different types.

Strands, Roles, and Strand Spaces.

Each protocol in this framework is a set of *parameterized roles*, and each role is a sequence of *actions*. Currently, there are only **send** and **receive** actions for passing messages among participants. An instance of a role with concrete parameters defines a *strand*, or a particular *run* of a particular *principal* in the protocol. As an example, consider the Needham-Schroeder public key authentication protocol [NS78] consisting of three messages:

$$\begin{array}{ll} A \to B: & \{N_A, A\}_{K_B} \\ B \to A: & \{N_A, N_B\}_{K_A} \\ A \to B: & \{N_B\}_{K_B}. \end{array}$$

The notation used here is a traditional way of writing security protocols: the first line states that a message $\{N_A, A\}_{K_B}$ (that is, a nonce generated by A and the name of the principal A encrypted with the principal B's public key) is sent by A to B. The nonce N_A in this case is a challenge to B, and A wants to verify that B can decrypt the message and extract the challenge, thereby proving its identity. In the second message, B packs the extracted N_A together with its own challenge N_B, encrypts it with A's public key, and sends it to A. Finally, A responds to the B's challenge, at which point both participants believe that their respective partner is indeed the one he claims to be, which is the goal of authentication. Additionally, they believe to share a common secret, N_A and N_B, since these are fresh nonces that were never sent out in clear (i.e. unencrypted).

In strand space model, this protocol can be represented as two roles, $Initiator$ and $Responder$, defining A and B respectively, parameterized by all the atomic messages that appear in the protocol. Actions for receiving a message are labeled by question mark, and sending is labeled with exclamation point:

$$
\begin{array}{cc}
\underline{Responder[A,\,B,\,N_A,\,N_B]} & \underline{Initiator[A,\,B,\,N_A,\,N_B]} \\
1 :?\{N_A,\,A\}_{K_B} & 1 :!\{N_A,\,A\}_{K_B} \\
\Downarrow & \Downarrow \\
2 :!\{N_A,\,N_B\}_{K_A} & 2 :?\{N_A,\,N_B\}_{K_A} \\
\Downarrow & \Downarrow \\
3 :?\{N_B\}_{K_B} & 3 :!\{N_B\}_{K_B}
\end{array}
$$

When instantiated with concrete messages, roles generate *strands*. Concrete protocol runs are only constructed with strands, that is, fully instantiated roles. Both roles and strands can be considered as graphs with actions being the nodes, and the connected arrows being the edges.

Under a normal run of the protocol, the Initiator's action 1 which sends the first message will be followed by the Responder's action 1 receiving that message, and so on. Connecting the corresponding sending and receiving nodes in the strands with another type of edges (singe arrow "$\rightarrow$", representing the *causal dependency*) results in a graph over the strands' nodes with two types of edges, which is called a *strand space*. The requirement on each causal dependency edge is that the adjacent nodes must have the same message (no message corruption in the transmission).

The Intruder Model.

The intruder in this model is not a single special entity possessing the network, as it is often done in other security models, but rather a gang of collaborating principals, each of which can do only one malicious thing at a time (snoop a message and resend it several times or simply drop it, decrypt a message and pass it on to someone else, etc.), but all together they are as powerful as any standard intruder can be. Each of the intruders, being just a principal, is also represented by a role. There are 7 intruder roles used in [SBP01][7]; we list them below with only a brief explanation. For conciseness, each role is represented as a sequence of actions in angle brackets; additionally, compound actions of the form $?(m_1 \,\mathbf{and}\, m_2)$ and $!(m_1 \,\mathbf{and}\, m_2)$ are added, meaning that both m_1 and m_2 must be received or sent, but the order of these atomic actions is unimportant.

- $M[t] = \langle !m \rangle$, where message m belongs to the initial knowledge of the intruder;
- $F[m] = \langle ?m \rangle$: receiving and dropping the message (useful for checking secrecy: if F receives a supposedly secret m, there is a problem with the protocol);

[7] Actually, there are 8, but two of them, K and M, are essentially the same: one sends an intruder's key, and the other — any other message originally known to the intruder.

- $T[m] = \langle ?m,\, !(m \,\mathbf{and}\, m)\rangle$: "tee" role, resend a message twice (duplicate);
- $V[m_1,\, m_2] = \langle ?(m_1 \,\mathbf{and}\, m_2),\, !(m_1,\, m_2)\rangle$: concatenates two messages;
- $R[m_1,\, m_2] = \langle ?(m_1,\, m_2),\, !(m_1 \,\mathbf{and}\, m_2)\rangle$: split the concatenation, it is the dual of V;
- $E[k,\, m] = \langle ?(k \,\mathbf{and}\, m),\, !\{m\}_k\rangle$: encrypts a message;
- $D[k,\, m] = \langle ?(k^{-1} \,\mathbf{and}\, \{m\}_k),\, !m\rangle$: decrypts a message (dual of E); here k^{-1} is a *decryption key* for messages encrypted with k (for instance, the corresponding private key, if a message is encrypted with a public key).

Bundles.

We say that a strand space represents a *complete* protocol run when all the receiving nodes are connected with corresponding sending nodes in the strand space, and for any node, all the nodes preceding it in its strand are also present in the strand space (that is, a strand space may include partial strands, but it must be backward-closed under the $\Rightarrow$ relation). If a strand space satisfies these conditions, it is called a *bundle*.

Specifications: the Logic and Example Properties.

Properties are expressed in a propositional logic that specifies which strands must or must not appear in any protocol execution. More specifically, the only atomic formulas are of the form $s \in C$, where s is a (possibly partial) strand, and C is a bundle, and each specification is given by

$$\forall C.\, f,$$

where f is a propositional formula over the atomic formulas in which the bundle symbol must be C. The meaning of a specification is that in any bundle (i.e. any complete protocol execution) there must always be certain combination of strands as specified by f.

For instance, a one-way authentication property of the Needham-Schroeder protocol can be expressed as follows:

$$\forall C.\, Responder[A,\, B,\, N_A,\, N_B] \in C \rightarrow Initiator[A,\, B,\, N_A,\, N_B] \in C.$$

That is, if an entire $Responder$ strand appears in a complete protocol run, then the complete $Initiator$ strand must also be in the same protocol run. This indeed expresses authentication, because the $Responder$ has successfully completed the protocol run and believes it is indeed talking to the corresponding instance of the $Initiator$ (notice that the parameters are the same). Otherwise, if this property does not hold, the $Responder$ must have been fooled by the intruder. Note, that the roles of the participants must be fully instantiated by some message constants in the formula, so they become strands.

Although the formula under the scope of $\forall C$ is an arbitrary propositional formula, it is sufficient to consider only formulas of the form $\forall C.\ \bigwedge \Phi \to \bigvee \Psi$, where Φ and Ψ are sets of atomic formulas.

Proof System: Semi-Bundle, Sequent, and Proof Rules.

The idea behind the proof system is the following. Given a specification $\forall C.\ \bigwedge \Phi \to \bigvee \Psi$, consider all bundles that contain strands mentioned in Φ and check that they all contain at least one strand mentioned in Ψ. Let us denote the corresponding sets of strands by Γ and Δ. Since there may be (and usually are) infinitely many such bundles, a compact representation for *sets of bundles* is used, called *semi-bundle*. We omit the formal definition of semi-bundle, but intuitively it is similar to a strand space, where all the nodes are backward-closed under the $\Rightarrow$ relation, and instead of the causal relation $\to$ another relation is used, which is roughly a transitive closure of $\to \cup \Rightarrow$ with all intermediate nodes dropped from the graph. It corresponds to finding the node which sends a message *for the first time*. Such a semi-bundle represents all bundles consistent with it in the message flow.

The verification process starts with a semi-bundle consisting of only the strands from Γ, and each step tries to *complete* the semi-bundle by finding a receiving node that is not connected to any sender (an *unbound goal*) and bind it to all possible sending nodes, one at a time, possibly adding new strands, thus, creating several refined versions of the semi-bundle. At each step we check whether the current semi-bundle has a strand from Δ, and if it does, the current subgoal is proven, since any bundle represented by the current semi-bundle has to contain that strand, and therefore, satisfies the original specification. When all goals are bound but no strand from Δ is present, the property has been violated and a counterexample is constructed (a complete bundle violating the specification, which is essentially an attack on the protocol).

The above procedure is formalized as a proof system. Since we only need the current semi-bundle l and the set of strands Δ, the sequent has the following form:[8]

$$l \vdash \Delta.$$

[8]Strictly speaking, the sequent contains the semi-bundle together with some additional information, but it is not essential in this informal description. See [SBP01] for details.

The two main inference rules implement, respectively, the successful termination condition:

$$\frac{}{l \vdash \Delta}\text{final} \qquad \text{if } l \cap \Delta \neq 0$$

and the refinement step:

$$\frac{l_1 \vdash \Delta \quad \ldots \quad l_n \vdash \Delta}{l \vdash \Delta}\text{split},$$

where $\{l_1, \ldots, l_n\} = \mathcal{F}(l, g)$, and $\mathcal{F}(l, g)$ generates all possible semi-bundles that bind a goal g originally unbound in l. The definition of $\mathcal{F}$ is rather complex and is omitted in this description. Intuitively, it finds all the nodes that may send the message received in g, and generates new semi-bundle for each of them with the appropriate binding, possibly adding new strands. If none of the two rules applies, the subgoal is unprovable and represents an attack on the protocol.

To connect the original form of the specification $\forall C. f$ with this "working" representation, another *initial* sequent is introduced:

$$P \vdash \forall C. f$$

where P is the protocol, with an additional rule:

$$\frac{l_1 \vdash \Delta \quad \ldots \quad l_k \vdash \Delta}{P \vdash \forall C. f}\text{init},$$

which translates the formula into a set of "working" subgoals as we outlined earlier.

In the actual implementation there are many optimizations which help prune large or even non-terminating proof subtrees. Some of these optimizations are expressed through *pruning theorems*, which check whether the set of bundles represented by the current semi-bundle is empty (and, therefore, the subgoal is vacuously true). A few such pruning theorems are mentioned in [SBP01], and they are implemented in *SyMP* and mentioned below.

4.3.2 Language Description

The input language is designed to specify multiple independent security protocols in the spirit of strand space representation. The capitalization of the keywords is unimportant (that is, `begin` is the same as `Begin` or `BEGIN`), but it is important for all the other names and identifiers (so, `Alice` is not the same as `alice`). A protocol declaration is of the form

```
protocol <id> ::=
  begin <definition> { [ ; ] <definition> } end
```

The list of definitions declares internal roles, predicates, and theorems for the protocol.

```
<definition> ::= <role> | <predicate> | <theorem>
```

A role has a name, a list of formal parameters, and a body consisting of the list of *actions*. Each action may have an optional *label*, and actions are separated by an optional semicolon to prevent potential ambiguity. There are only two built-in actions `send` and `receive` that send and receive messages to and from the environment. The intended recipient or a sender is not specified when sending/receiving a message, since under the assumption that the intruder has a complete control over the network, this information is not of any use.

```
<role> ::= role <id> '[' <params> ']' =
                  begin <action> {[ ; ] <action>} end
<action> ::= [ <id> : ] send <message>
           | [ <id> : ] receive <message>
<role_params> ::= <role_param_block> {; <role_param_block> }
<role_param_block> ::= <id> {, <id>} : <typeSpec>
<typeSpec> ::= new <type> | fresh <type> | unique <type>
             | Self | FreshNonce
<type> ::= Message | Principal | Nonce | <type> { * <type> }
         | Encrypted `[' <type> `]' | Key
         | PrivKey | PubKey | SymKey
```

The keywords `new` and `fresh` are exact synonyms, and they specify that the value of the corresponding parameters will be freshly generated by each strand (or *instance*) of this role, and their values initially are not accessible to any other strand. The keywords `self` and `freshNonce` are synonymous to `new principal` and `new nonce` respectively. The freshness rule for the parameters of type `self` (or `new principal`) is slightly different from all the other types. These parameters define the *identities* of each strand. A strand may have several identities at once. The identities themselves are always public knowledge, but the private keys and shared symmetric keys are initially accessible only to those strands that own the corresponding identities. Unlike the other types of "fresh" parameters, the same "seft" parameter can be used by different roles, with some special technical restrictions on the penetrator strands. This corresponds to one principal playing different roles at the same time.

The keyword `unique` specifies that the role checks this parameter for uniqueness in all of its runs. That is, if the same role is played by the principal with exactly the same identities (`Self` parameters), then the value of all `unique` parameters must be different in all such strands. The unique parameters are not necessarily generated by the role itself, and in fact, they are often generated by others, and the role only checks if it has already seen the value before. This mechanism helps to rule out some replay attacks on the protocols.

In the strand space model, messages are not sent to any participant in particular, but instead are "posted" to the common network for everyone to see. It is expected that the intended recipients will find their messages themselves. At the same time this allows the intruder to tamper with the messages as it wishes, and the goal of the verification is to show that the intruder still cannot break the protocol w.r.t. the properties stated as theorems.

All the objects in the language are explicitly typed, and the types have a well-defined *subtyping hierarchy*. The most general type is `Message`, which is a direct supertype of `Principal`, `Nonce`, `Key`, `Encrypted[`τ`]` (the type of an encrypted message of type τ), and a tuple type $\tau_1 * \cdots * \tau_n$ (the type of the concatenation of n messages of the respective types).

`Principal` and `Nonce` types do not have any subtypes.

`Key` has 3 immediate subtypes: `PrivKey`, `PubKey`, and `SymKey` for private, public, and symmetric key types respectively.

Subtypes of `Encrypted[`τ_1`]` can only be other types of the form `Encrypted[`τ_2`]`, where τ_2 is a subtype of τ_1.

Tuple type $\tau = \tau_1 * \cdots * \tau_n$ can only have tuple subtypes of the same length, and its components must be subtypes of the respective components of τ.

When parameters are instantiated, the type of each formal parameter must be a supertype of the actual parameter.

Messages in the `send` and `receive` actions are formed from atomic messages by concatenation and encryption operators. Also, keys are extracted from principals that own them with `PK` (public key), `PVK` (private key) and `SymKey` (shared symmetric key) operators. An inverse of a key can be constructed using `INV k` operator. The inverse of a symmetric key is the key itself, and for a private/public keys their inverse is the opposite key in the pair.

```
<message> ::= <id> | ( <message> { , <message> } )
            | '{' <message> '}' <message> | INV <message>
            | PK <message> | PVK <message>
            | SymKey(<message>, <message>)
```

The atomic messages are formal parameters (there are no static or user-defined constants). Concatenation is a tuple of messages, and encryption $\{m\}k$ takes a message to encrypt and a key k of some `Key` (sub)type. A decryption operation is simply an encryption with the inverse key for private/public key ciphers. But more often, decryption is done implicitly by *pattern matching* on messages in the receive action. For example, if a role has an action

```
receive {N, A} (PK B)
```

and B is the owner of the role (parameter of type Self), then the message can be implicitly decrypted by using N and A in the later actions of the role.

Properties of the protocols are specified with predicates and theorems:

```
<predicate> ::= predicate <id> ’[’ <params> ’]’ = <formula>
<theorem> ::= theorem <id> ’[’ <params> ’]’ = <formula>
<params> ::= <param_block> {; <param_block> }
<param_block> ::= <id> {, <id>} : <type>
```

Predicates and theorems have exactly the same syntax except for the initial keyword. Their purposes, however, are different. Predicates assign names to formulas to be used later in other predicates and theorems, and serve as convenient macro definitions. Theorems define formulas that must always be true for the given protocol, and this is what the proof system will try to prove. Although allowed by the input language, it probably does not make much sense to use a theorem name as a macro for its formula in other predicates and theorems, since once proven, it will always be true. We leave this option in the language for the future extensions when we will be able to use previously proven theorems in the proofs of the other theorems.

Notice, that the parameters to predicates and theorems are of the same form as to the roles, except that we do not allow to specify freshness.

A <formula> is a propositional logic formula over the atomic propositions. An atomic proposition is a set of (possibly partial) strands from one of the roles in the protocol or the intruder’s roles. A strand can be either an instance of a role (then it is a complete strand), or a substrand defined with the help of the labels on the actions:

```
<formula> ::= <atomic_formula> | not <formula>
            | <formula> <op> <formula>
<op> ::= and | ‘&’ | or | implies | ’->’ | iff | ’<->’
<atomic_formula> ::= <strand> | ’{’ <strand> {, <strand>} ’}’
<strand> ::= <role_inst> | <role_inst> ’.’ <id>
           | <role_inst> ’.’ <id> ’-’ <id>
<role_inst> ::= <id> ’[’ <message> {, <message>} ’]’
```

For instance, {Sender[A,B], Receiver[A,B].start-finish} is a set of two strands; the first is a complete strand instantiated from the role Sender, and the second is a partial strand consisting of actions of the `Receiver`'s instance starting from the action labeled `start` and ending with the action labeled `finish`, inclusive. For concrete examples, see Figure 4.7 and the Appendix B.

The formula is interpreted over complete (possibly infinite) executions of the protocol. An execution in this case is a (possibly infinite) set of strands with `send` and `receive` actions connected with *causal relation* to form the *strand space* (see Section 4.3.1 for the discussion of these terms). An execution is complete (or, it forms a *bundle*) if every `receive` action is connected to some `send` action in the strand space, and every strand contains its initial action, as defined by the corresponding role. An atomic proposition is true in a complete execution (bundle) if its strand appears in the bundle, possibly as a substrand of another strand. The entire formula is true iff it is true in every possible complete execution of the protocol. A predicate or theorem is true in the protocol, if its formula is true for all possible values of its parameters. For more formal definitions please refer to [SBP01].

4.3.3 The Proof Rules and Commands: Running *Athena* in *SyMP*

The *Athena* module has 3 the most important proof rules:

init processes the theorem in the initial sequent and generates the "working" sequents $l \vdash \Delta$ from it;

final completes the proof of the sequent if the current state l has all the nodes of at least one strand in Δ; and

split picks and binds a goal in all possible ways. It takes an optional parameter to specify the goal manually.

The pruning theorem related to the encryption depth is hard-coded into the `split` rule. The other pruning theorems are related to the availability of keys and are implemented as two rules:

prune_keys: If the protocol never sends keys of certain types, and there is a goal requesting such a key, then the state is contradictory, and the proof of this sequent is completed immediately.

prune_signed: If the protocol never sends messages signed with principal's private keys, and there is a goal requesting a signed message, then this goal is unsatisfiable, and the rule proves the current sequent.

```
protocol NSPL =
begin
 role Init [NA: FreshNonce; NB: Nonce; A: Self; B: Princi-
pal] =
  begin
   start:  send {(NA, A)} (PK B) receive {(NA, NB, B)} (PK A)
   finish: send {NB} (PK B)
  end

role Resp [NA: Nonce; NB: FreshNonce; A : Princi-
pal; B : Self] =
  begin
   start:   receive {(NA, A)} (PK B)
   respond: send {(NA, NB, B)} (PK A)
   finish:  receive {NB} (PK B)
  end

 predicate responded[NA, NB: Nonce; A, B: Principal] =
  Resp[NA,NB,A,B]

 predicate initiated[NA, NB: Nonce; A, B: Principal] =
  Init[NA,NB,A,B].finish

 theorem agreement[NA, NB: Nonce; A, B: Principal] =
  (initiated[NA,NB,A,B] -> Resp[NA,NB,A,B].respond) and
  (responded[NA,NB,A,B] -> initiated[NA, NB, A, B])
end
```

Figure 4.7: Needham-Schröder authentication protocol with the Lowe's fix in *Athena* language.

Proof System-Specific Commands

Some information about the sequent is not printed by default. You can view that information using the proof system commands:

explain prints the protocol run in a supposedly human-readable format; that is, one line per message, each message of the form

$$A \to B_1, \ldots, B_n : \text{Message},$$

along with the list of participants and interm constraints, if there are any. The reason there may be multiple recipients is because the intruder may duplicate the message and forward it to anyone he wants. The true reason, however, is that each line is really all bindings from the same term to all its goals. This command is mighty handy, and you may find yourself not using any other commands at all.

bindings prints the list of goal bindings and the current *interm constraints* on the terms. In *Athena*, a goal g can be bound by a term t that has more components than the goal needs. If such a term t is a variable of a `Message` type, the exact components of t are unknown, and the only information we have is that g is an interm of t. This information is recorded in the state and is used in later goal bindings.

print_debug prints the complete information about the current sequent: the goal bindings, the interm constraints, the nodes with all the information stored in them (strand ID, role ID, all the parameters, and the corresponding action), and some other information about the protocol used by the pruning theorems.

Automatic Proof Search

The *SyMP* prover has a general purpose strategies that can be used in *Athena* to search for proofs automatically (see the general prover manual). The suggested way of using strategies is by letting *Athena* execute all the available rules in a particular order, namely, the `final` rule and the rules for pruning theorems (to cut the proof as early as possible), then the `split` rule, and, finally, the `init` rule. The reason the `init` rule is the last is because it will never be applicable after the initial sequent is transformed into a "state" sequent, so you don't want to try it every time just to see it fail. But the other rules will not apply to the initial sequent, and if you run the strategy from the very first sequent, the prover will eventually try the `init` rule. This will happen only at the beginning and, possibly, when *Athena* stumbles upon a counterexample, that is, at most twice in the entire proof, so the overhead of trying so many rules before `init` is negligible.

To summarize, the super-duper *Athena* strategy is:

```
(repeat (try final prune_keys prune_signed split init)),
```

where `try` is a strategy for trying each rule in the list until one applies, and the `repeat` strategy applies the rule or strategy in its argument repeatedly as long as it applies, generating the proof subtree in the depth-first manner.

Chapter 5

Experimental Results

Formal verification is in many ways an experimental science, and this work is not an exception. Having a sound underlying theory and even a rather general implementation of it in the *SyMP* tool is still not sufficient to demonstrate its practical applicability.

This chapter describes our experience with applying our methodology to several concrete verification examples in two problem domains: hardware and security protocols. The main purpose of the examples is to illustrate the versatility of our new methodology, that is, its applicability to a wide variety of problem domains.

Even though we expriment with only two problem domains, the experiments are still quite conclusive. First of all, the two domains are extremely different from each other, and yet our methodology is equally applicable to both of them. This suggests that it is also likely to work well for many other not so vastly different problem domains. In addition, one of the specializations, the default proof system, is a very general framework by itself. It is possible to consider much more narrow specializations based on it, which may significantly boost the expected practical performance.

The default proof system, which is a specialization of theorem proving to model checking, is illustrated most thoroughly by the verification of a simplified version of the IBM Cache Coherence protocol (Section 5.1). It is verified in two ways, demostrating that the default proof system is expressive enough to handle both theorem proving-oriented reasoning, and also the reasoning very specific to model checking. This alone is a good indication that such a proof system is a practical integration of model checking and theorem proving. A few more examples of larger sizes are provided to explore the scalability of the approach.

The second proof system, *Athena*, is highly specialized to reason about security protocols. The suite of examples in Section 5.3 demonstrates the practical performance of our methodology on this very different problem domain. Again, there are no extremely complicated cases (and in fact, it is very difficult to find a protocol on the appropriate level of abstraction that *Athena* would not tackle with ease), and the main goal of these experiments was to get an idea about the practical performance of the methodology in this problem domain.

The degree of automation in *Athena* is much higher that that of the default proof system, and one reason for that is a higher degree of specialization. This supports our conjecture that the more specialized the verification tool is, the more automated the verification process can be made for comparable complexity of the examples.

5.1 IBM Cache Coherence Protocol

As a very simplified version of the protocol, we consider the following configuration. The entire system consists of N identical nodes (or *clients*) representing *processor units*, each with its own cache, and each cache having only one single *cache line*. In addition, there is a special *home node* which keeps track of the location of the data and ensures coherence. We assume that each node has a direct connection with the home node, and the communication is done through *virtual channels*. Each node has 3 virtual channels connecting it to the home node, each of which has a one-element queue buffer (the actual protocol allows larger queues).

Each cache line can be in one of the 3 states: *Invalid*, *Shared*, and *Exclusive*. 'Invalid' state means the cache line is empty, 'Shared' means it has a valid read-only copy, and 'Exclusive' means it has the only copy of the data in the entire system, and it is a read-write copy. We will completely ignore the data stored in the cache line and concentrate only on the cache's state. The property we want to verify is a *mutual exclusion* property. It is an invariant that, whenever some cache is in the 'Exclusive' state, all the other caches must be in the 'Invalid' state. The home node keeps the list of nodes that currently have a shared or exclusive copy of the data (SharerList), the node ID of the node it is currently servicing (CurrentNode), the command it is about to send or receive on the bus (CurrentCommand), and a boolean flag indicating whether anyone has an exclusive copy (ExclusiveGranted). Whenever a node wants to obtain a copy of the data, it sends a request to the home node through its $channel_1$ indicating whether it wants a shared or exclusive copy. The home node then picks one of the clients among the requesters and starts servicing it. If the client requests a shared copy, the home node checks that there is no exclusive copy issued and sends the shared response to the client. If there is a node with the exclusive copy, then the home node first sends an *invalidate request* to that client through *channel*$_{2\_4}$, which prompts the client to write back the data and switch to the 'Invalid' state. The home node then waits for the acknowledgement of this write-back on *channel*$_3$, and after that sends the response back to the requesting client on *channel*$_{2\_4}$. Similarly, if a client requests an exclusive copy, the home node checks if anybody else is holding the data and sends the *invalidate* request to all of them. This time both shared and exclusive states are important. After all clients send their confirmations of invalidation, the home node issues a response to the original client allowing it to get an exclusive copy. Initially we assume that all caches are in the 'Invalid' state, and the state of the home node is initialized accordingly.

The protocol is *asynchronous*, meaning that all transactions are serialized and performed one-by-one. In this example, the transition relation is defined using a *guarded nondeterministic choice* assignment of the form:

```
choose (cl: Client):
      <guard1> => asst1
   |  <guard2> => asst2
        ...
endchoose
```

The `choose` operator first picks nondeterministically a value of the parameter `cl` which ranges over all clients (or caches) that makes at least one guard true; then it nondeterministically picks one true guard and executes the corresponding assignment. Each such assignment in the example corresponds to an atomic transaction of the protocol; for example, a cache places a message in its channel, or reads a message and changes its state. If none of the guards can be made true for any value of `cl`, then a special `nop` assignment is executed which keeps the state of the relevant variables unchanged.

For this example, we provide two versions of its proof: one is typical for traditional theorem proving, and the other one heavily uses ideas from model checking.

5.1.1 An Approach Biased to Theorem Proving: Induction on Time

The mutual exclusion property that we want to prove is a *safety property*, or an *invariant*, the one that must hold in every reachable state of the system. One way to prove it is by *induction on time*. The base of the induction ensures that it holds in all the initial states. The induction step consists of proving the property at time $t + 1$ under the assumption that it holds at time t. Note that in order for the induction to go through, the property itself must be *inductive*, that is, it must be strong enough to capture all necessary assumptions for the time t, but not too strong to hold at $t + 1$ under these assumptions. Since not every invariant is inductive, in order to prove an invariant ϕ we may have to *strengthen* it to obtain another invariant ψ which (a) is inductive, and (b) implies our original invariant ϕ in *every* reachable state of the model. To formalize this induction principle in our proof system, we introduce a proof rule induct_AG:

$$\frac{M;\, \Sigma;\, \Gamma \vdash \Delta,\, \psi \quad M' = (S, \rightarrow, S);\, \Sigma,\, \psi;\, \cdot \vdash \phi \quad M';\, \Sigma;\, \psi \vdash \mathbf{AX}\,\psi}{M = (S, \rightarrow, I);\, \Sigma;\, \Gamma \vdash \Delta,\, \mathbf{AG}\,\phi}\,\mathsf{induct\_AG}.$$

Here M' is the same model as M except that the set of initial states is now the entire set S, so the last two premisses require that the property hold in all the states satisfying Σ, the assumed set of invariants of M. Recall, that we only need to prove ψ true in all states *reachable* from I in the original model M, not in *all* the states S. The set Σ may have some important properties that all the reachable states satisfy, and without which the induction might not go through. Since Γ and Δ are only relevant to the initial states of the current model, when the set of initial states changes, Γ and Δ must be removed to preserve soundness, as is done in the last two premisses. However, Σ does not have to be removed, since it is not bound to the initial states, and this is why we need it as a separate field in our sequent. Later we will show how formulas can be added to Σ and used.

Back to our example, we formulate our cache mutual exclusion property as the following first-order CTL formula:

Theorem 5.1.1.

$$\mathbf{AG}\,\phi \equiv \mathbf{AG}\,\forall c_1,\, c_2.\,(c_1 \neq c_2 \wedge \mathrm{cache}(c_1) = \mathrm{Exclusive} \rightarrow \mathrm{cache}(c_2) = \mathrm{Invalid}).$$

This corresponds to a sequent:

$$M;\, \cdot;\, \cdot \vdash \mathbf{AG}\,\phi.$$

Let us for the beginning choose $\psi = \phi$ in the hope that the property is already inductive and try to verify it starting with the induct_AG rule. After we apply the rule, we end up with 3 new *subgoals* to prove. The second subgoal:

$$M' = (S, \rightarrow, S);\, \phi;\, \cdot \vdash \phi$$

is trivially true, since the only conclusion is the same as the assumption. It is easy to see that the first subgoal

$$M;\, \cdot;\, \cdot \vdash \phi \tag{5.1}$$

is also true simply because all the caches are initially in the 'Invalid' state. To prove this formally, we first eliminate the universal quantifier with the *Skolemization* rule:

$$\frac{M;\,\Sigma;\,\Gamma \vdash \Delta,\, A(a)}{M;\,\Sigma;\,\Gamma \vdash \Delta,\, \forall x.\, A(x)}\forall^a_R.$$

Here a is a new *Skolem constant* that does not appear anywhere below in the proof. The soundness of the rule is guaranteed by the fact that this new constant is new and *uninterpreted*, so if we prove the formula $A(a)$ above the line for all the interpretations of a, this implies that $\forall x.\, A(x)$ also holds by definition of the $\forall$-quantifier.

After applying the $\forall^a_R$ rule twice to (5.1) to generate two Skolem constants for c_1 and c_2, we obtain

$$M;\,\cdot;\,\cdot \vdash (a_1 \neq a_2) \wedge \mathrm{cache}(a_1) = \mathrm{Exclusive} \to \mathrm{cache}(a_2) = \mathrm{Invalid}. \tag{5.2}$$

Next, we need a set of propositional rules to eliminate the implication and conjunction:

$$\frac{M;\,\Sigma;\,\Gamma,\, A \vdash \Delta,\, B}{M;\,\Sigma;\,\Gamma \vdash \Delta,\, A \to B}\to_R \qquad \frac{M;\,\Sigma;\,\Gamma,\, A,\, B \vdash \Delta}{M;\,\Sigma;\,\Gamma,\, A \wedge B \vdash \Delta}\wedge_L$$

which yield

$$M;\,\cdot;\, a_1 \neq a_2,\, \mathrm{cache}(a_1) = \mathrm{Exclusive} \vdash \mathrm{cache}(a_2) = \mathrm{Invalid}.$$

Notice, that the two assumptions are not really needed in this case, so we use a *weakening* rule to remove them; this will help the *cone of influence* rule to reduce the model to a smaller one:

$$\frac{M;\,\Sigma;\,\Gamma \vdash \Delta}{M;\,\Sigma;\,\Gamma,\, A \vdash \Delta}W_L \qquad \frac{M|_V;\,\Sigma;\,\Gamma \vdash \Delta \quad V = \mathrm{COI}(M,\, \Sigma \cup \Gamma \cup \Delta)}{M;\,\Sigma;\,\Gamma \vdash \Delta}\mathrm{COI}.$$

Here $\mathrm{V} = \mathrm{COI}(M,\, \Phi)$ is a function that computes the set of variables V which may influence any of the specifications in the set Φ, and $M|_V$ is a *slice* of the model M that has only variables from V, and all the other variables are removed. In our case, the only relevant variable is $\mathrm{cache}(a_2)$, which does not depend on any other variables in the initial state, and therefore, the set V in the 'COI' rule consists of only this variable. So, our new subgoal becomes

$$M|_{\{\mathrm{cache}(a_2)\}};\,\cdot;\,\cdot \vdash \mathrm{cache}(a_2) = \mathrm{Invalid}. \tag{5.3}$$

Note, that the traditional cone of influence reduction would not reduce M just to one variable, since $\mathrm{cache}(a_2)$ depends on many other variables in the *next state assignment*. However, our formula is a propositional one, and therefore, the next state assignment will never be used, so we ignore it in the computation of V.

Finally, since the subgoal (5.3) has a finite model and only propositional specifications, we can apply *model checking* rule to it directly, which easily proves the sequent:

$$\frac{\mathrm{ModelCheck}(M,\ \bigwedge\Sigma,\ \bigwedge\Gamma \to \bigvee\Delta) = \mathbf{true}}{M;\ \Sigma;\ \Gamma \vdash \Delta}\mathrm{MC},$$

where $\mathrm{ModelCheck}(M,\ \sigma\ \gamma)$ is a function that runs a model checker on the model M restricted to the global invariant σ, and returns $\mathbf{true}$ if the property γ holds in the model.

This completes the proof of the first two subgoals of the induct_AG rule. Now we tackle the most difficult third subgoal representing the induction step:

$$M;\ \cdot;\ \phi \vdash \mathbf{AX}\,\phi. \tag{5.4}$$

The proof of (5.4) requires looking at the transition relation of the model M. Recall, that the transition relation of M is represented by the choose statement, a nondeterministic choice among guarded assignments. Our formula refers only to the current and the next state (and not any further that that), and we have to prove the formula ϕ on the right for *every* next state (because it is under the $\mathbf{AX}$ operator). Since the only way to get to the next state is to execute one of the branches of `choose`, we have to consider all possible cases present in the statement. A new rule corresponding to such a case split is the following:

$$\frac{\begin{array}{c}M_1;\ \Sigma;\ \Gamma,\ g_1(a) \vdash \Delta \quad \cdots \quad M_n;\ \Sigma;\ \Gamma,\ g_n(a) \vdash \Delta \\ M_{\mathrm{nop}};\ \Sigma;\ \Gamma,\ \forall x.\neg\bigvee_{i=1}^{n} g_i(x) \vdash \Delta\end{array}}{M;\ \Sigma;\ \Gamma \vdash \Delta}\mathrm{choose}^a,$$

where M contains an assignment

```
choose x:
     g1(x) => asst1(x)
        ...
   | gn(x) => asstn(x)
endchoose
```

and M_i is the same as M except that the entire `choose` statement has been replaced with $\mathtt{asst}_i(a)$; similarly, M_{nop} has `choose` replaced with the `nop` assignment; a is a new *Skolem constant*, similar to the one in the $\forall^a_R$ rule, which represents the nondeterministic choice of the value of x made by the choose statement. Since an assignment i can only be chosen if its guard is true, the guard is added to the list of assumptions Γ in each case. And in the last case, when no guard can be made true, the corresponding formula is also added to the assumptions.

Observe, that the rules cone and choose^a aggressively exploit the particular representation of the model specific to our proof system (and the problem domain), and the fact that the model is cleanly separated from the specification. These rules would be extremely hard and cumbersome to implement efficiently in traditional theorem proving; and the fully general version of choose^a presented here is impossible to incorporate into the finite-state framework of classical model checking.

In our example, there are 10 branches in the `choose` statement, therefore, 11 cases are generated by the rule $\mathrm{choose}^{\mathrm{a}}$. Some of the assignments do not modify any of the caches, and therefore, their proofs should consist of a cone of influence reduction and model checking. However, the sequent (5.4) still refers to all of the caches, and since the number of caches is an uninterpreted parameter, the resulting model will not be directly suitable for model checking even after the cone of influence reduction. So, first we have to eliminate the quantifiers in ϕ on both side of (5.4). The $\forall$-quantifier on the right hand side is in the scope of the $\mathbf{AX}$ operator, and the rule $\forall^a_R$ is not directly applicable to it. However, $\mathbf{AX}$ and $\forall$ commute, and we can pull the quantifier to the top level:

$$\frac{M;\, \Sigma;\, \Gamma \,\vdash\, \Delta,\, \forall x.\, \mathbf{AX}\, A(x)}{M;\, \Sigma;\, \Gamma \,\vdash\, \Delta,\, \mathbf{AX}\, \forall x.\, A(x)} \forall \mathbf{AX}_R.$$

After applying this rule to $\mathbf{AX}\,\phi$ in any of the remaining subgoals, we can then eliminate the quantifier by the $\forall^a_R$ rule. In fact, since this has to be done in every subgoal, we can backtrack in our proof and apply $\forall \mathbf{AX}_R$ and $\forall^a_R$ before the choose^a rule. This leaves us with 11 subgoals of the form

$$M_i;\, \cdot;\, g_i(a),\, \phi \,\vdash\, \mathbf{AX}\,((a_1 \neq a_2) \wedge \mathrm{cache}(a_1) = \mathrm{Exclusive} \rightarrow \mathrm{cache}(a_2) = \mathrm{Invalid}).$$

The universal quantifier in ϕ on the left can be eliminated (or *instantiated*) with the $\forall_L$ rule:

$$\frac{M;\, \Sigma;\, \Gamma,\, A(t) \,\vdash\, \Delta}{M;\, \Sigma;\, \Gamma,\, \forall x.\, A(x) \,\vdash\, \Delta} \forall_L,$$

where t is an arbitrary term. Notice, that the quantifiers are treated differently on the left hand side, since formulas in Γ are assumptions that we use to prove formulas in Δ. An assumption of the form $\forall x.\, A(x)$ means that for any value of x, $A(x)$ is assumed to be true; therefore, in particular, it is true for any term t that we might need this assumption for.

For those subgoals corresponding to the cases of `choose` that do not modify the states of the caches it is natural to instantiate c_1 and c_2 in ϕ on the left with a_1 and a_2 respectively. The guard g_i in any of these subgoals is irrelevant, and we can safely remove it from the sequent with the W_L rule. After that, the cone rule finds that the only state variables relevant to the formulas in the sequent are $\mathrm{cache}(a_1)$ and $\mathrm{cache}(a_2)$. However, even though the model seems to be finite, direct model checking is still not applicable, since we do not have an interpretation for a_1 and a_2. In fact, we do not even know exactly how many state variables we have, since when $a_1 = a_2$, the model has only one state variable $\mathrm{cache}(a_1)$ (which is the same as $\mathrm{cache}(a_2)$), and two variables otherwise. This is due to the fact that a_1 and a_2 are Skolem constants, and we have to prove the sequent for all possible their interpretations. However, no matter how we interpret them, the only property we can deduce about them is whether they are equal or not, since $a_1 \neq a_2$ is the only predicate that references them. Also, this is the only fact that is needed to determine the aliasing of the state variables. So, our proof has to split the cases on whether $a_1 = a_2$ or not:

$$\frac{M;\,\Sigma;\,\Gamma,\,A \vdash \Delta \quad M;\,\Sigma;\,\Gamma \vdash \Delta,\,A}{M;\,\Sigma;\,\Gamma \vdash \Delta}\mathrm{cut}.$$

The cut rule is a general rule for splitting cases on whether an arbitrary formula A is true or not, and since for any interpretation either A or $\neg A$ must hold, proving the sequent for both cases is sufficient for the original sequent without any mention of A to hold.

Although, theoretically, the cut rule is exactly what we need at this step, to be able to use the additional hypothesis $a_1 = a_2$ or its negation we would have to introduce more rules. Instead, we introduce a more specialized but more powerful rule that does all the required transformations in one shot:

$$\frac{M_{\sigma_1};\,\Sigma_{\sigma_1};\,\Gamma_{\sigma_1} \vdash \Delta_{\sigma_1} \quad \cdots \quad M_{\sigma_N};\,\Sigma_{\sigma_N};\,\Gamma_{\sigma_N} \vdash \Delta_{\sigma_N}}{M;\,\Sigma;\,\Gamma \vdash \Delta}\mathrm{abstractSplit}.$$

In this rule, we collect all the Skolem constants used in the original sequent, both in the model and in the formulas, and construct all different sets of assumptions σ_i about the relationships among these Skolem constants (N is the number of different σ_i's). These assumptions, in the simplest case, are equalities or disequalities between each pair of Skolem constants. This rule is precise (or *invertible*) when Skolem constants are only compared for equality among each other or used as array indices. This condition could be easily weakened to allow Skolem constants be compared with some concrete constants (e.g., if a term $a = 0$ appears in a formula), and even used in some predicates other than the equality, but this is beyond the scope if this example. For our purposes it is sufficient to view the set of assumptions σ_i as an *equality relation* over Skolem constants defined by its *equivalence classes*: $\sigma_i = \{S^i_1, \ldots, S^i_{k_i}\}$. For each equivalence relation σ_i we modify the model appropriately (merge all the state variables $x(a_1)$ and $x(a_2)$ such that a_1 and a_2 are in the same equivalence class of σ_i) and evaluate all the predicates of the form $a_1 = a_2$ to either $\mathbf{true}$ or $\mathbf{false}$, depending on whether a_1 and a_2 are in the same equivalence class or not.

After this transformations for each σ_i, we obtain new sequents in which different Skolem constants are guaranteed to have only different interpretations, and can be replaced with *abstract constants* (so that further rules would not confuse them with unreduced Skolem constants). Thus, abstract constants are essentially Skolem constants, except that different abstract constants are guaranteed to have different interpretations, and $b_1 = b_2$ iff b_1 and b_2 are one and the same constant.

In our example, for those subgoals (cases of the `choose` statement) that do not modify any cache variables, after deleting the guard from the assumption we end up with only 2 Skolem constants in the sequent: a_1 and a_2. There are two possible equivalence relations: one where $a_1 = a_2$, and the other where $a_1 \neq a_2$. Thus, for each such sequent we obtain 2 new subgoals with 1 and 2 state variables respectively ($\mathrm{cache}(a_1)$ and $\mathrm{cache}(a_2)$), and these sequents can be easily discharged by model checking (the MC rule).

Some other sequents do modify cache variables and may need to use the guard as an assumption; in this case the Skolem constant a that comes from the application of the rule choose^a has to be counted in the $\mathrm{abstractSplit}$ rule, yielding 5 new subgoals, each of which are again ready to be model checked. There are, however, two subgoals for which the model checker will produce a counterexample: these are the cases when a cache line receives a shared and exclusive copy of the data respectively. Since our invariant ϕ allows one of the caches to be exclusive at any time, and does not say which one it must be as long as all the others are invalid, making another cache shared or exclusive violates the invariant. This indicates that ϕ is either not an invariant, or is not an *inductive invariant*. Recall, that we are trying to prove the original $\mathbf{AG}$ -property by induction on time, and any induction requires an inductive invariant to go through. Since we still hope that ϕ is actually an invariant (and the designers keep assuring us), let us assume that ϕ is just not strong enough and try to *strengthen* it. Since we have introduced the invariant in the very first rule $\mathbf{AG}_R$, we have to kill the entire proof and redo it from scratch starting from the $\mathbf{AG}_R$ rule, only this time giving it a better invariant than ϕ.

If we look at the counterexamples, we see that the home node grants a cache shared or exclusive privileges only if it believes that no other cache is in the exclusive state, which is determined by the state of its SharerList and ExclusiveGranted state variables. Therefore, we also need to prove that these two state variables are always in sync with the actual states of the caches; so we add another conjunct to the invariant ϕ, obtaining a new, stronger, invariant:

$$\begin{aligned}\phi \quad \wedge \quad & \forall x.\,(\mathrm{cache}(x) = \mathrm{Exclusive} \quad \rightarrow \quad \mathsf{ExclusiveGranted}) \\ \wedge \quad & \forall y.\,(\mathrm{cache}(y) \in \{\mathrm{Shared},\ \mathrm{Exclusive}\} \quad \rightarrow \quad \mathsf{SharerList}(y)).\end{aligned}$$

Although this formulation is already good enough, for the sake of the proof's simplicity we rewrite this invariant slightly to have all the quantifiers on the top level:

$$\begin{aligned}\psi \equiv \forall c_1,\, c_2. \quad & (c_1 \neq c_2 \wedge \mathrm{cache}(c_1) = \mathrm{Exclusive} \quad \rightarrow \quad \mathrm{cache}(c_2) = \mathrm{Invalid}) \\ \wedge \quad & (\mathrm{cache}(c_1) = \mathrm{Exclusive} \quad \rightarrow \quad \mathsf{ExclusiveGranted}) \\ \wedge \quad & (\mathrm{cache}(c_1) \in \{\mathrm{Shared},\, \mathrm{Exclusive}\} \quad \rightarrow \quad \mathsf{SharerList}(c_1)).\end{aligned}$$

For brevity, we denote the three subformulas by A_ϕ, A_1, and A_2 respectively:

$$\psi \equiv \forall c_1,\, c_2.\, A_\phi(c_1,\, c_2) \wedge A_1(c_1) \wedge A_2(c_1).$$

Let us now quickly go over the proof once again and see if we can complete it this time. As before, we apply the induct_AG rule to the original formula $\mathbf{AG}\,\phi$, but this time using our new invariant ψ instead of ϕ. We obtain 3 new subgoals:

$$M;\, \cdot;\, \cdot \vdash \psi \tag{5.5}$$

$$M';\, \psi;\, \cdot \vdash \phi \tag{5.6}$$

$$M';\, \cdot;\, \psi \vdash \mathbf{AX}\,\psi. \tag{5.7}$$

The first subgoal (5.5) can be split into 3 subcases by the $\wedge_R$ rule:

$$\frac{M;\, \Sigma;\, \Gamma \vdash \Delta,\, A_1 \quad M;\, \Sigma;\, \Gamma \vdash \Delta,\, A_2}{M;\, \Sigma;\, \Gamma \vdash \Delta,\, A_1 \wedge A_2}\wedge_R,$$

and then each of the cases is proven by $\forall^a_R$, cone, and MC rules (Skolemization, cone of influence, and model checking), since all the caches are initially invalid. The second subgoal (5.6) is proven by Skolemizing the RHS ($\forall^a_R$) and instantiating the LHS with the new Skolem constants ($\forall_L$), and then "flattening" the result with the $\wedge_L$ rule, which yields two identical formulas in Γ and Δ and completes its prove. Again, the most interesting part is the induction step (5.7). As before, we first apply all the *invertible* rules to the sequent (the "precise" rules, such that the sequent holds iff all the premisses of the rule hold), which are the $\forall\mathbf{AX}_R$ rule (swapping $\mathbf{AX}$ and $\forall$) and Skolemization $\forall^a_R$, yielding

$$M';\, \cdot;\, \psi \vdash \mathbf{AX}\,(A_\phi(a_1,\, a_2) \wedge A_1(a_1) \wedge A_2(a_1)).$$

Applying the choose^a rule to split on the nondeterministic choice in the model M' again gives us similar 11 subgoals which we deal with by instantiating the quantifiers in ψ on the left, possibly removing the guard, applying the abstractSplit rule and model checking. This time the two problematic cases, where the cache variables are modified, go through. However, since the property we want to prove became stronger, some other cases that we proved earlier fail this time. In particular, in the case when an element of the SharerList is reset because the home node receives an invalidate acknowledgement from the corresponding cache line, we did not keep track of whether the cache indeed switched to the 'Invalid' state. Therefore, assuming that the cache "cheated" leads to violation of the A_2 part of the invariant. The counterexample prompts us for another strengthening of the invariant, and we have to do another iteration of the proof.

Such an iterative process of strengthening the invariant should eventually yield an inductive invariant that allows us to prove the property. Even in our relatively small example, after spending several hours adding new conjuncts to the invariant and redoing the proof we still could not find an explicit inductive invariant. Larger examples may need significantly more complex invariants, and doing the proof over and over again could be tedious (even in our example it was not easy), and figuring out how to strengthen the invariant when we get a counterexample may require significant manual effort and expertise in the design of M. On the other hand, since we want to verify a parameterized design by induction on time, there is little hope that we can get by without constructing an inductive invariant one way or the other.

Let us now step back and think about the designer's intentions in this protocol. First, we notice that the home node enforces the mutual exclusion property by keeping track of the caches' states in its own variables SharerList and ExclusiveGranted. The real state of the cache must be consistent with the home node's perception, although it may be delayed in time because of the communication latency. This view of the protocol prompts for a different verification strategy, where we first prove that the home node indeed enforces mutual exclusion of *its local view* of the caches, and then show that the local view has enough relevance with the actual state of the caches to prove our main property.

Let us define a function homeCache(c) that computes the state of a cache c as perceived by the home node:

```
homeCache(c) = if sharerList(c) then
                  if exclusiveGranted then Exclusive
                  else Shared endif
               else Invalid endif
```

and formulate the *home mutual exclusion* property as a lemma:

Lemma 5.1.2.

$$\mathbf{AG}\,(\forall c_1,\, c_2 :\; c_1 \neq c_2 \wedge \mathsf{homeCache}(c_1) = \text{Exclusive} \rightarrow \mathsf{homeCache}(c_2) = \text{Invalid}).$$

This invariant turns out to be inductive, and moreover, it only depends on the state of the home node; the states of the communication channels and caches are irrelevant. We can exploit this fact by abstracting away the communication channels from the model (that is, making all the channel variables completely nondeterministic), after which the cone of influence will remove the caches from the model, since they become disconnected from the home node and cannot influence its behavior.

Formally, we prove this lemma by induction on time, as before. The induct_AG rule will produce three subgoals:

$$M;\,\cdot;\,\cdot\,\vdash\phi_h,$$

$$M' = (S,\,\to,\,S);\,\phi_h;\,\cdot\,\vdash\phi_h,\quad\text{and}$$

$$M';\,\cdot;\,\phi_h\,\vdash\mathbf{AX}\,\phi_h.$$

The second subgoal is trivially true (the inductive invariant implies the invariant we want to prove, which is the same in this case). The first subgoal is proven by Skolemizing ($\forall^a_R$ rule), flattenning ($\to_R$ and $\wedge_L$ rules), deleting the formulas on the LHS (W_L rule). The cone of influence reduction on the resulting sequent:

$$M;\,\cdot;\,\cdot\,\vdash\mathrm{homeCache}(a_2) = \mathrm{Invalid},$$

leaves only two boolean variables in the model: $\mathsf{sharerList}(a_2)$ and $\mathsf{exclusiveGraned}$, and then the MC rule completes the proof of this subgoal.

Finally, in the most interesting third subgoal, the universal quantifier on the right is pulled out of the $\mathbf{AX}$ operator (rule $\forall\mathbf{AX}_R$), and quantifiers are eliminated by $\forall^a_R$ and $\forall_L$ (the quantified variables on the LHS are instantiated with the corresponding Skolem constants from the RHS). At this point, we could simply apply the $\mathrm{abstractSplit}$ rule, which would produce two finite-state subgoals ready to be model checked. However, to increase the efficiency of model checker, we will also abstract away the caches and communication channels, as we mentioned above. A corresponding rule is the following:

$$\frac{M';\,\Sigma;\,\Gamma\,\vdash\,\Delta}{M;\,\Sigma;\,\Gamma\,\vdash\,\Delta}\mathrm{abstractVar},$$

where $M' = (S',\,\to',\,I')$ is the same as $M = (S,\,\to,\,I)$, except that some state variables are removed and replaced by nondeterministic choices everywhere in M. The formulas in the sequent are restricted to ACTL in Δ and ECTL in Σ and Γ. In our case, we use this rule to abstract all caches and communication channels from the system. This leaves at most 3 boolean state variables in the cone of influence of the resulting model after all the abstractions: $\mathsf{sharerList}(a_1)$, $\mathsf{sharerList}(a_2)$, and $\mathsf{exclusiveGranted}$, which can be easily model checked.

Next, we need to connect the $\mathrm{homeCache}$ values to the real caches. Observe that the home node can only see the states of the caches through the communication channels, and will maintain the cache's state in its local variables according to the cache's state *together with the state of its channels*. In other words, it seems natural to view the cache together with its communication channels as a single *abstract cache*. We define another function $\mathrm{absCache}(c)$ that maps a cache with channels to a cache state by a table:

channel3	channel2_4	cache	exclusiveGranted	absCache
InvalidateAck	$\emptyset$	Invalid	false	Shared
InvalidateAck	$\emptyset$	Invalid	true	Exclusive
$\emptyset$	$\emptyset$	*state*	–	*state*
$\emptyset$	Invalidate	Shared	false	Shared
$\emptyset$	Invalidate	Exclusive	true	Exclusive
$\emptyset$	grantShared	$\neq$Exclusive	–	Shared
$\emptyset$	grantExclusive	Invalid	–	Exclusive
Otherwise				Error

Notice that we also use $\mathrm{absCache}$ to rule out some configurations that should not be possible to achieve by mapping them to an "Error" state. For instance, the protocol should never send an "Invalidate" message to a cache that is already invalid, and the cache should be in the "Invalid" state when it sends an "InvalidateAck."

Our next lemma states that the home node indeed maintains its state consistent with the current state of $\mathrm{absCache}$:

Lemma 5.1.3.

$$\mathbf{AG}\,\mathrm{absCache} = \mathrm{homeCache}.$$

Notice, that we stated this lemma as an equality between functions; this is equivalent to stating the equivalence for each individual cache:

$$\mathbf{AG}\,\forall c.\,\mathrm{absCache}(c) = \mathrm{homeCache}(c).$$

Before we start proving this lemma, it is worth making sure that it will be helpful in the proof of our main mutual exclusion property, and this is the next step we want to check. First, from Lemmas 5.1.2 and 5.1.3 we can conclude that the $\mathrm{absCache}$ also enjoys the mutual exclusion property:

Lemma 5.1.4.

$$\mathbf{AG}\,\forall c_1,\, c_2.\, c_1 \neq c_2 \wedge \mathrm{absCache}(c_1) = \mathrm{Exclusive} \rightarrow \mathrm{absCache}(c_2) = \mathrm{Invalid}.$$

To prove this formally in our proof system, we need a rule that can use a lemma stated earlier:

$$\frac{M;\, \Sigma;\, \Gamma,\, \psi \vdash \Delta}{M;\, \Sigma;\, \Gamma \vdash \Delta}\mathrm{Lemma},$$

where there is a lemma $M;\, \cdot;\, \cdot\, \vdash \psi$ stated *before* the current lemma in the specification for *the same model* M. An alternative is to use the cut rule to introduce the lemma from within the proof. Next, we have to replace $\mathrm{homeCache}$ with $\mathrm{absCache}$ on the right. A rule that takes an equality from the assumptions and substitutes its RHS expression for its LHS in the rest of the sequent is a reasonably general form of a proof rule that we can use here:

$$\frac{M[A/B];\, \Sigma[A/B],\, A = B;\, \Gamma[A/B] \vdash \Delta[A/B]}{M;\, \Sigma,\, A = B;\, \Gamma \vdash \Delta}\mathrm{replace}.$$

Notice, that the equiality has to be in the invariant part of the sequent. If it were in Γ, then it would only be assumed to hold in the initial states, and we would not be able to replace it soundly inside the $\mathbf{AG}$ operator. Next, we need a way to move an assumption from Γ (where the lemma is originally placed) to Σ, which can be achieved with the Cut_I rule:

$$\frac{M;\, \Sigma,\, \phi;\, \Gamma \vdash \Delta \quad M;\, \Sigma;\, \Gamma \vdash \Delta,\, \mathbf{AG}\,\phi}{M;\, \Sigma;\, \Gamma \vdash \Delta}\mathrm{Cut}_I,$$

where the second premiss is proven by the 'Lemma' rule. Let us introduce a more convenient rule for the frequent special case of lemmas in the form of $\mathbf{AG}\,\phi$:

$$\frac{M;\, \Sigma,\, \psi;\, \Gamma \vdash \Delta}{M;\, \Sigma;\, \Gamma \vdash \Delta}\mathrm{Lemma}_{\mathbf{AG}},$$

where a lemma of the form $M;\, \cdot;\, \cdot\, \vdash \mathbf{AG}\,\psi$ is present in the specification before the current one with the same model M. To summarize, applying the rules $\mathrm{Lemma}_{\mathbf{AG}}$ (for both Lemmas 5.1.2 and 5.1.3) and $\mathrm{replace}$ proves Lemma 5.1.4. Another invariant that we need is that abstract cache never reaches the "Error" state:

Lemma 5.1.5.

$$\mathbf{AG}\,\forall c.\, \mathrm{absCache}(c) \neq \mathrm{Error}.$$

This lemma can be proven by using Lemma 5.1.2, replacing $\mathrm{absCache}$ with $\mathrm{homeCache}$, and checking that $\mathrm{homeCache}$ function can never yield "Error." The last step can be done, e.g. by expanding the definition of $\mathrm{homeCache}$, lifting the **if**-expressions over the disequality and evaluating the resulting expression.

It is easy to see now that Lemmas 5.1.4 and 5.1.5 together imply our main mutual exclusion property (Theorem 5.1.1), since $\mathrm{cache}(c) = \mathrm{Exclusive}$ implies $\mathrm{absCache}(c) = \mathrm{Exclusive}$, and $\mathrm{absCache}(c) = \mathrm{Invalid}$ implies $\mathrm{cache}(c) = \mathrm{Invalid}$ under the assumption that $\mathrm{absCache}(c) \neq \mathrm{Error}$.

So, the only missing part of the proof of Theorem 5.1.1 is Lemma 5.1.3. However, trying to prove Lemma 5.1.3 directly quickly shows that this property is not inductive, and we need to strengthen the invariant. To cut the story short, the final inductive invariant that eventually allowed us to prove Lemma 5.1.3 is the following:

$$\forall c.\, \mathrm{homeCache}(c) = \mathrm{absCache}(c) \wedge \mathrm{enabledRules} \wedge \mathrm{phaseInvar},$$

where $\mathrm{enabledRules}$ and $\mathrm{phaseInvar}$ are predicates over an additional *monitor state variable* 'phase', which keeps track of what phase of the protocol the home node is in. Specifically, phase ranges over $\{\mathrm{Initial}, \mathrm{Invalidating}, \mathrm{Responding}\}$. The $\mathrm{Initial}$ phase means the home node is not servicing any of the clients, and is ready to accept new requests. The $\mathrm{Invalidating}$ phase is when the chosen client is requesting an exclusive copy of the data, but someone else is holding the data, and the home node is in the process of invalidating those cache lines. Finally, the $\mathrm{Responding}$ phase is when the request is being granted to the client.

The predicate $\mathrm{enabledRules}$ defines which rules can be enabled in each phase of the protocol, that is, which actions the system can perform in a given phase. If the home node can accept new requests, then it must be in the $\mathrm{Initial}$ phase. If any action relevant to invalidating a cache is enabled, then the phase must be $\mathrm{Invalidating}$. If the home node can grant shared or exclusive privileges to some client, then the phase is $\mathrm{Responding}$.

The $\mathrm{phaseInvar}$ predicate adds additional invariants to each of the phase. In the $\mathrm{Initial}$ phase, no pending grant or invalidate messages should remain in the channel queues, and home node must not be servicing any requests. In the $\mathrm{Invalidating}$ phase, the set of clients to be invalidated must be a subset of clients holding the data, and each to-be-invalidated client must satisfy the following: it should be either in a non-invalid state and the $\mathrm{Invalidate}$ message should not have been sent to it yet, or there is a message pending to it that grants it read or write privileges. Additionally, the home node must be servicing some request (not being idle), and if it is a $\mathrm{reqShared}$ request, then some client must have been granted the exclusive copy before. The $\mathrm{Responding}$ phase requires that the list of to-be-invalidated clients be a subset of those holding the data (even though no invalidation is already necessary), no client is receiving the $\mathrm{Invalidate}$ message, no client is responding with the invalidate acknowledgment, and if the serviced request is $\mathrm{reqExclusive}$, then no client should hold the data. On top of that, the home node must be servicing some request (not being idle), and if the request is $\mathrm{reqShared}$, then no one should have been granted the exclusive copy before.

5.1.2 An Approach Biased to Model Checking: Abstraction and Induction without Inductive Invariant

It should be pretty clear by now that the complexity of the inductive invariant itself, not to mention the proof using it, is the primary bottleneck in the verification by direct induction on time. Another way to attack the verification problem of a parameterized system is to try an aggressive abstraction at the very beginning with the hope to obtain a detailed enough but finite abstraction and simply model check the resulting model. The inductive invariant in this case is implicitly (and automatically) constructed by the model checker and is completely hidden from the user. The complexity in this case is shifted to figuring out the right abstraction.

In the previous section we have introduced a rule $\mathrm{abstractSplit}$, which provides an easy way to build abstract models. This rule basically assumes that only the constants $a_1, a_2, \ldots$ explicitly mentioned in the specification are important in the proof. It collects all the state variables that have common types with these constants and abstracts the types of those variables to $\{\perp, a_1, a_2, \ldots\}$, where $\perp$ stands for all the values other than the constants. For instance, if we want to leave only $\mathrm{cache}(a_1)$ and $\mathrm{cache}(a_2)$ in the model, the potentially infinite type of the cache index '`Client`' can be abstracted to just 3 values: $\{a_1, a_2, \perp\}$. This immediately makes the types finite, and the model may become small enough for a model checker to handle. A version of this technique based on symmetry reduction has first been introduced by Ken McMillan [McM98].

Instead of trying to prove Theorem 5.1.1 by induction on time, let us try to apply the abstraction and model check the property directly. First, we lift the universal quantifier from inside the scope of $\mathbf{AG}$ using the rule:

$$\frac{M;\,\Sigma;\,\Gamma \vdash \forall x.\,\mathbf{AG}\,\phi}{M;\,\Sigma;\,\Gamma \vdash \mathbf{AG}\,\forall x.\,\phi}\forall\mathrm{lift}_{\mathbf{AG}}\,.$$

After Skolemization (rule $\forall^a_R$) and abstraction of the `Client` type ($\mathrm{abstractSplit}$ rule), we obtain two finite abstract models in which this type is $\{a_1, a_2, \perp\}$ and $\{a, \perp\}$, for the cases when the Skolem constants a_1 and a_2 are different and the same respectively.

Model checking the property in the first model, however, yields a (spurious) counterexample: since we abstract away all the caches indexed by $\perp$, they are allowed to send any messages at any time nondeterministically. In particular, one such cache sends an

$\mathrm{InvalidateAck}$ while the home node is waiting for it, but before the cache that actually holds the data has a chance even to receive the $\mathrm{Invalidate}$ request (which stays in its channel queue). The home node then happily grants an exclusive copy to another cache, thus violating the mutual exclusion property.

This counterexample suggests that we should keep track of whether the "other" caches have an exclusive copy or not, and therefore, whether they can send such a bogus message. One solution (suggested to us by Ken McMillan) is to introduce an additional lemma that does exactly that:

Lemma 5.1.6.

$$\mathbf{AG}\,(\mathrm{homeExclusiveGranted} \wedge \mathrm{channel}(c) = \mathrm{InvalidateAck} \to c = \mathrm{owner}).$$

Here c is the state variable that indicates which of the asyncronous components is making progress at the current step, and owner is another *monitor state variable* that keeps track of which cache holds the exclusive copy of the data. Adding the lemma as a restriction on the abstracted model is sufficient to prove Theorem 5.1.1 by direct model checking, and all that is left is to prove Lemma 5.1.6.

It is important to stress once again that Lemma 5.1.6 was not magically pulled out of a hat, but rather was derived directly from the counterexample to the main Theorem 5.1.1. Understanding the reasons for the counterexample involves some human insight. In our case, the reason is that a cache that clearly does not own an exclusive copy of the data nevertheless sends the $\mathrm{InvalidateAck}$ message, which breaks the protocol. Therefore, to prevent this from happenning, we introduce a lemma stating that only the current owner is allowed to send this message. A new state variable is added to keep track of the currect owner. Since this variable only "monitors" the behavior of the original system and does not interfere with it in any way, this change to the model does not affect the validity of the original specification.

In more involved examples, this approach of "patching holes" in the initial abstraction with additional lemmas may have to be repeated several times and may go several levels deep (lemmas may need additional lemmas for their proofs), since a spurious counterexamples may be generated for other yet undiscovered reasons. Luckily, in this example we only have one such "hole," and Lemma 5.1.6 successfully "patches" it and allows the proof of Theorem 5.1.1 to go through.

The first obvious attempt at proving the lemma is to try the same "abstract and model check" approach, except that we do not have any Skolem constants to base our abstraction on. However, we have two state variables mentioned in the lemma that have potentially infinite types: c and owner. We then use the equality

$$\phi \equiv \forall x.\, \zeta = x \to \phi,$$

where ζ is an arbitrary term, to rewrite the lemma in the following way:

$$\begin{aligned}&\mathbf{AG}\,(\forall a_1,\, a_2.\, c = a_1 \wedge \mathrm{owner} = a_2\\&\quad \to \mathrm{homeExclusiveGranted} \wedge \mathrm{channel}(c) = \mathrm{InvalidateAck}\\&\qquad \to c = \mathrm{owner}).\end{aligned}$$

This now becomes familiar, and we swap $\forall$ and $\mathbf{AG}$, abstract the type of c and owner to $\{a_1,\, a_2,\, \perp\}$ and $\{a,\, \perp\}$ (remember, that we have to generate two cases here), and run the model checker on the two abstract models. But this time the abstract model does not satisfy the property, and we are again left with a counterexample, similarly to our experience with the main theorem.

Of course, one solution could be just the same: find another lemma that helps us eliminate the counterexample, and which we, hopefully, can prove. This chain of lemmas, however, may go on too far, and we may eventually end up with an invariant similar to the one in the previous section.

Another solution is to fall back to the induction on time and hope that it will be easier for this (seemingly) simpler lemma. The hope here is quite thin though, since even in a "pure" theorem proving environment it is not too hard to derive the main mutual exclusion property form this lemma, and most likely, all the complexity of the proof will be shifted into the proof of this lemma.

Fortunately, there is a middle-ground solution, also first used by McMillan [McM98], however, with a different argumentation. Our version of this solution is the following sequence of transformations:

$$\begin{aligned}\mathbf{AG}\,\phi &\equiv \mathbf{A}[\neg\phi\,\mathbf{R}\,\phi]\\&\equiv \mathbf{A}[\neg\phi\,\mathbf{R}\,\forall x.\, \zeta = x \to \phi]\\&\equiv \forall x.\, \mathbf{A}[\neg\phi\,\mathbf{R}\,\zeta = x \to \phi],\end{aligned}$$

where $\mathbf{A}[a\,\mathbf{R}\,b]$ is the *release* operator, the dual of *until*. That is,

$$\neg\mathbf{A}[a\,\mathbf{R}\,b] \equiv \mathbf{E}[\neg a\,\mathbf{U}\,\neg b].$$

We omit the formal proof of each step of the transformation for brevity, but it is not very difficult to reconstruct. The key observation here is that the final property after Skolemization and abstraction is weaker than the property after similar transformations directly on the $\mathbf{AG}$ formula, and this turns out to be sufficient to prove the lemma by direct model checking. The term ζ in our case is the state variable c in Lemma 5.1.6.

Another way to see this transformation is as a *generalized induction on time* : assuming that the property holds up to time $t-1$ (as opposed to *at* time $t-1$), prove that *in the set of reachable states of the abstract model* the property holds at time t. The meaning of the formula $\mathbf{A}[\neg\phi\,\mathbf{R}\,\phi]$ is exactly that: for each path, either ϕ holds at all times along the path, or there is a point where $\neg\phi$ becomes true, and ϕ must hold up to (and including) that state. Of course, if $\neg\phi$ becomes true, then ϕ cannot be true in that same state, and the formula will be violated. However, it allows us to express the assumption that ϕ holds all the time up to $t-1$, and in order for the formula to hold, it must also be true at time t. Lifting the universal quantifier from the right-hand side to the top level corresponds to proving ϕ at time t case by case (for each value of the term ζ), but under the assumption that the entire ϕ holds at $t-1$. This is why it is a weaker property (i.e. more likely to hold) than $\forall x.\,\mathbf{AG}\,(\zeta = x \to \phi)$, since the latter can be considered as a generalized induction that assumes $\zeta = x \to \phi$ up to time $t-1$, and not the full ϕ.

5.1.3 Summary

Despite the clear difference in the complexity of the proof, we will not suggest that one method is clearly better than the other based on just this example. Instead, we would like to emphasize that both approaches can be realized within our unified framework without any loss of generality or any significant loss of efficiency, compared to the specialized tools like *PVS* or Cadence *SMV* in which this example has also been verified. This suggests that our framework is indeed a balanced and general enough combination of model checking and theorem proving. In addition, it is quite possible that other more complex examples may require extensive use of both model checking and theorem proving techniques, in which case neither *PVS* nor *SMV* may have enough expressive power or efficiency, but our tool will still be able to handle such examples.

5.2 Floating Point Unit Controller

A research group at Saarland University has designed a complete microprocessor in *PVS* language, formally verified it all the way to the gate level, and implemented on a Xilinx FPGA. This is a superscalar microprocessor with out-of-order instruction scheduling and an instruction set typical to any general purpose microprocessor. In particular, it supports

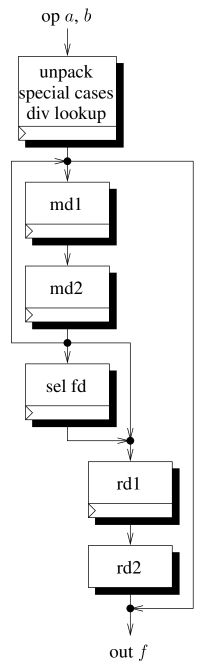

Figure 5.1: Floating Point Unit Controller

floating point arithmetic with single and double precision, and is fully IEEE compliant. The multiplication and division operations are performed in a designated functional unit whose design and verification are described in [BI01]. The unit is pipelined with a shared multiplication circuit used in both multiplication and division operations, and the nature of this specific pipeline allows the unit to complete instructions out of order.

The pipeline is driven by the controller whose architecture is shown in Figure 5.1. When an instruction is loaded into the unit, its operands are unpacked and checked for special cases (division or multiplication by 0, `INF`, or `NaN`), and then passed to the actual processing stages, except for the special cases where the result is computed and returned directly by the unpacker. The processing stages consist of two-staged multiplier (md1 and md2), selector of the resulting significand (select fd), and two rounding stages (rd1 and rd2), which also normalize the result to conform to the IEEE standard.

The multiplier stages are used both for the multiplication and for computing the next approximation in the iterative division algorithm. Since the multiplier is pipelined, when a division instruction is dispatched into the unit before a multiplication, the multiplication will leave the unit after passing through the pipeline once, but the division will be fed back to the pipeline for another 2 or 3 iterations, depending on the precision of the operands. The result is, the multiplication instruction will leave the unit before the division, even though it has arrived later.

The verification task is to prove that the underlying algorithm produces an accurate and IEEE compliant result, and that the controller drives the pipeline in such a way that parallel instructions do not interact and are not being lost or duplicated.

In this experiment we have only verified the latter property, that is, the correctness of the pipeline control signals. The data correctness and, in particular, IEEE compliance have not been verified in our tool *SyMP*.

The entire floating point unit operates within an out-of-order module of the microprocessor driven by the Tomasulo's scheduler. To avoid hazards related to false data dependences (write after read and write after write), the values to be written back to the register file are copied into the reorder buffer and tagged with a unique *tag*. When the floating point unit loads several instructions simultaneously, even if they complete out of order, it will still be possible to identify the result by the tag.

One of the most important properties of the controller is to ensure that it does not lose or duplicate the tags, provided that no duplicate instructions were loaded into the unit.

We have taken the model from the original *SMV* description (the authors have verified it both in *SMV* and *PVS*), translated it into *SyMP*, parameterized it by the tag width, and verified the general parameterized model. The *SyMP* proof corresponds very closely to the proof in *SMV*, where the arbitrary tag width is modeled with uninterpreted types.

The verification itself was very straighforward, the entire proof of the most complex property (preservation of tags) consists of only 8 simple steps that run within a couple of seconds through the first 7 rules (the state reduction techniques producing a finite model), and the only computationally demanding step is the run of the model checker (CMU version of *SMV*), which takes about 30 minutes on a 200MHz Pentium processor using 122MB of RAM. Most of the running time is spent on dynamic variable reordering without which *SMV* runs out of memory. Since the generated *SMV* model is not very efficient in the current version of *SyMP*, we expect that the run time can be significantly reduced by using better optimizations in the translation phase of the `modelcheck` inference rule.

5.3 *Athena*: Experiments in Security Protocol Verification

The purpose of building and experimenting with the *Athena* proof system in the context of this thesis is to demonstrate that *SyMP* is a versatile prover generator and can handle problem domains that are as far from hardware verification as security protocols. Also, it has been an instructive experience on how *SyMP* helps in designing new proof systems, understanding the specific subtleties of the problem domain, and developing new heuristics and methodologies for (automatic or manual) proof search, extensions to the specification language, etc.

The original verification algorithm *Athena* was created by Dawn Song [SBP01, Son99], who based her work on the *strand space model* by Thayer Fábrega, Herzog and Guttman [THG99]. A big part of the implementation of the *Athena* module in *SyMP* was done by Alex Groce, who designed most of the input language, the typechecker, and the infrastructure for the inference rules. Our contribution is in completing the implementation of the inference rules, extending the language and the typechecker with more features, and performing the experiments.

We have verified several well-known protocols taken from the survey by Clark and Jacob [CJ97]. Most of the protocols can be verified or refuted completely automatically, and some of them required manual guidance, but we have not found a protocol where no automatic proof search was possible at all.

Our complexity measure of the verification will be the number of *steps* it took the prover to find a flaw or prove the protocol secure, and the number of *states* generated during the proof search. A proof step is just an application of a proof rule. A state is a synonym for *sequent* in this context, and the reason we call them states is to follow the terminology in the original *Athena* paper [SBP01], where a state is a compact representation of a (possibly infinite) family of protocol runs. If a protocol is proven correct, then a rule must have been applied to each state (or sequent), and the two numbers will be identical. But in case of a proof failure, the sequent that cannot be reduced represents an attack on the protocol, and other sequents in yet unexplored branches of the proof need not be proven or disproven, so the number of states can be larger than the number of the proof steps.

Each proof step generally takes no more that a tenth of a second to apply, and in an automatic mode, a 50-step proof can be found in a matter of a few seconds. The memory required to store each state is negligible for small proofs and only matters when several thousand states are generated; in such cases 128MB of our workstation was getting a bit tight. Fortunately, we have never encountered such proofs after implementing a simple heuristic for *goal binding* in the *split* rule. The largest proof we have found has 551 states, which is still well within the capacity of the machine.

The complete source code for all of the examples of this section is given in Appendix B.

5.3.1 Needham-Schroeder Authentication Protocols

Public Key Protocol.

In our first experiment, we used the *public key authentication protocol* by Needham and Schroeder [NS78] which in its simplest version consists of three messages:

$$\begin{aligned} A \to B:&\quad \{N_A,\, A\}_{K_B} \\ B \to A:&\quad \{N_A,\, N_B\}_{K_A} \\ A \to B:&\quad \{N_B\}_{K_B}. \end{aligned}$$

We use the standard notation for describing security protocols as above. Each line represents a message sent from one participant to another; $A \to B : M$ means A sends a message M to B. Messages encrypted with (public or symmetric) keys are written as $\{M\}_K$.

The goal of this protocol is to *authenticate* the principals A and B to each other. That is, by the end of a successful protocol run, A believes she is talking to someone possessing B's private key (and therefore, it must indeed be B, assuming private keys never get compromised), and B believes he is indeed talking to A. In addition, the two principals share a common secret, the two nonces N_A and N_B, from which they may derive a new symmetric session key for later communication.

Our model assumes that the encryption is "perfect," or unbreakable, which formally can be defined as

$$\{M_1\}_{K_1} = \{M_2\}_{K_2} \text{ iff } M_1 = M_2 \text{ and } K_1 = K_2.$$

Nonces (random *n*umbers used "*once*") generated by principals are denoted by the letter N subscripted with the principal's name (e.g. N_A is a nonce generated by A).

This protocol has a famous flaw first discovered by Gavin Lowe [Low95][1] and requires two parallel sessions of the protocol:

$$\begin{aligned} A \to Z:&\quad \{N_A,\, A\}_{K_Z} \\ Z(A) \to B:&\quad \{N_A,\, A\}_{K_B} \\ B \to Z(A):&\quad \{N_A,\, N_B\}_{K_A} \\ Z \to A:&\quad \{N_A,\, N_B\}_{K_A} \\ A \to Z:&\quad \{N_B\}_{K_Z} \\ Z(A) \to B:&\quad \{N_B\}_{K_B}. \end{aligned}$$

Here Z represents an attacker, and $Z(A)$ is the same attacker masquerading as A. We assume that the attacker has a complete control over the network and can intercept, drop,

[1] The attack was found only 17 years after the protocol has been published!

duplicate, and rearrange messages as he wishes, but he cannot randomly guess any secret data and cannot decrypt encrypted messages without knowing the decryption key.

The attacker uses the fact that the second message from the protocol does not bear any information of who generated it and for what purpose, and A happily accepts this message thinking that Z has honestly generated a new nonce N_B. In reality, A is being used an an *oracle* to decrypt the nonce N_B, so that the attacker can return it back to B, thus convincing him that he is talking to A, even though he is actually talking to Z, and A might have never heard of B before.

Athena finds this attack completely automatically in 36 steps, generating 44 states. The entire proof search in a "cold run" (that is, without any explicit or implicit help from the user) takes less than 3 seconds.

Gavin Lowe proposes a fix by including the responder's identity into the second message [Low95], so this attack now becomes impossible:

$$\begin{aligned} A \to B &: \ \{N_A,\, A\}_{K_B} \\ B \to A &: \ \{N_A,\, N_B,\, B\}_{K_A} \\ A \to B &: \ \{N_B\}_{K_B}. \end{aligned}$$

Athena proves that this protocol is indeed secure completely automatically in 27 steps.

Symmetric Key Protocol.

This version of the protocol is published in the same work by Needham and Schroeder [NS78], and is also found to be flawed, as described by Clark and Jacob [CJ97]. The protocol relies on the trusted server S for generating a *fresh* session key K_{ab} that must be securely distributed to the participants A and B. The shared keys K_{AS} and K_{BS} between the server and the corresponding principals are assumed to be known to the respective participants and not compromised. After receiving the session key, the participants authenticate to each other and are ready to communicate securely using that key, which constitutes the main purpose of the protocol:

$$\begin{aligned} A \to S &: \ A,\, B,\, N_A \\ S \to A &: \ \{N_A,\, B,\, K_{ab},\, \{K_{ab},\, A\}_{K_{BS}}\}_{K_{AS}} \\ A \to B &: \ \{K_{ab},\, A\}_{K_{BS}} \\ B \to A &: \ \{N_B\}_{K_{ab}} \\ A \to B &: \ \{N_B - 1\}_{K_{ab}}. \end{aligned}$$

The flaw described in [CJ97] requires a compromised session key from an old run of the protocol, and an old message 3 recorded by the intruder. This simple replay attack results in

B having to accept an old compromised key and believing that he is talking to A, whereas he would be talking to the intruder. To find this attack in *Athena*, we had to augment the role of the principal B by an extra message at the end of the protocol:

$$B \to Z : K_{ab},$$

that is, to compromise the key after the session is supposedly completed by all of the principals. The automatic proof search did not terminate, and we had to guide the prover for the first few steps before running the automated strategy that finally produced the attack after total of 68 steps, generating 164 states.

Besides the freshness attack, this protocol also suffers from a more serious flaw found by our *Athena* automatically (7 steps/31 states for one version, and 14 steps/48 states for the other). This attack comes in two flavors, neither of which requires any key to be compromised. The only assumptions we make is that the server S may also act as one of the participants (like A and B). To make it clearer who is playing which role, we adopt the following notation. The name of a *role* will be the long name of a participant (e.g. *Alice* is the role of A, etc.). The name of a participant by itself means that this participant plays its default role (for instance, $S \to A : M$ mean S acting as a *Server* sends M to A acting as *Alice*). When a participant plays some other role, we include this role in parentheses next to the principal's name (e.g. $S(Alice)$ is S playing the role of *Alice*). Similarly, as we have already done so earlier, Z will denote the intruder acting as such, and $Z(A)$ or $Z(S(Alice))$ is the intruder pretending to be the corresponding principal acting in a specified role.

The simpler version of the attack goes as follows:

$$\begin{aligned}
A \to S : &\quad A,\, B,\, N_A \\
S \to A : &\quad \{N_A,\, B,\, K_{ab},\, \{K_{ab},\, A\}_{K_{BS}}\}_{K_{AS}} \\
A \to Z(B) : &\quad \{K_{ab},\, A\}_{K_{BS}} \\
Z(A) \to S(\mathit{Bob}) : &\quad \{K_{ab},\, A\}_{K_{BS}} \\
S(\mathit{Bob}) \to A : &\quad \{N_S,\}_{K_{ab}} \\
A \to Z(B) : &\quad \{N_S - 1\}_{K_{ab}} \\
Z(A) \to S(\mathit{Bob}) : &\quad \{N_S - 1\}_{K_{ab}}.
\end{aligned}$$

The attacker uses the fact that the encryption is symmetric, and for a message encrypted with a symmetric key K_{BS} it is not possible to tell who of the two principals (S or B) has created the message for whom. This allows the attacker to intercept and forward the message addressed to B back to S as if generated by B acting as a server (who might also be a a legitimate backup authority) forwarded by A. The principal S then interprets this message as if A wants to talk to him and replies according to the protocol's role "*Bob*." As it can be seen, both A and S successfully complete their protocol sessions, and therefore, S believes he is talking to A (and he actually does, but he also thinks that the session key is unique for his session, which is not), and A believes she is talking to B, whereas B never even appears in the protocol at all, and might well be long dead.

Another version of this attack assumes that A can act as another server and is not required to be alive, and the omnipresent intruder Z manages to confuse everyone else around again.

$$\begin{array}{lrl}
(1) & S(Alice) \to Z(A(Server)): & S,\, B,\, N_S \\
(2) & Z(A) \to S: & A,\, B,\, N_S \\
(3) & S \to Z(A): & \{N_S,\, B,\, K_{ab},\, \{K_{ab},\, A\}_{K_{BS}}\}_{K_{AS}} \\
(4) & Z(A(Server)) \to S(Alice): & \{N_S,\, B,\, K_{ab},\, \{K_{ab},\, A\}_{K_{BS}}\}_{K_{AS}} \\
(5) & S(Alice) \to B: & \{K_{ab},\, A\}_{K_{BS}} \\
(6) & B \to Z(A): & \{N_B\}_{K_{ab}} \\
(7) & Z(B) \to S(Alice): & \{N_B\}_{K_{ab}} \\
(8) & S(Alice) \to B: & \{N_B - 1\}_{Kab}.
\end{array}$$

Here S wants to play the *Alice*'s role and talk to B. He decides to use A as a server (perhaps, B wouldn't trust S in generating the key by himself), but Z intercepts the message, tempers with it (since it is sent in clear) and resends it back to S as if from A who also wants to talk to B. The server S, being a faithful guy, generates a new session key and forms the required message (3), which the clever intruder returns right back to S but as if the reply from the second server A. Since S does not know he can decrypt the last part of the message and detect the fraud (he thinks it is encrypted with the key he does not have), he simply forwards it to B, unsuspecting. B now discovers that A wants to talk to him, generates a fresh nonce, and sends it to A as a challenge. His message is intercepted by the intruder and redirected to S, who retrieves the nonce and returns it back to B. This time the intruder does not even have to do anything, S has completed his evil mission for him in confusing B that he is talking to A, while, as we all know, A is long dead, and he is really talking to S. Although S is not as badly confused as B, he is also deceived into believing that B knows who he's talking to.

These two attacks were found by our implementation of *Athena* completely automatically, and to our complete surprise. We were not aware of these attacks before the experiment, and, to the best of our knowledge, they have not appeared in the literature. Dawn Song has found similar attacks with her implementation of *Athena*, but she has not published them yet, so these attacks can in fact be new, yet unknown attacks on the protocol.

5.3.2 Parallel Session Attack on a Simple Protocol

A simple intentionally flawed one-way authentication protocol from [CJ97]:

$$\begin{aligned} A \to B: &\quad \{N_A\}_{K_{AB}} \\ B \to A: &\quad \{N_A + 1\}_{K_{AB}} \end{aligned}$$

was found to be flawed by *Athena* in 4 steps, generating 7 states. The attacker forwards the first message back to A who now plays *Bob*'s role and responds with the message that she expects herself in the first session. The intruder, of course, is happy to forward the second message back to A, leaving her doubly confused:

$$\begin{aligned} A \to Z(B): &\quad \{N_A\}_{K_{AB}} \\ Z(B(\mathit{Alice})) \to A(\mathit{Bob}): &\quad \{N_A\}_{K_{AB}} \\ A(\mathit{Bob}) \to Z(B(\mathit{Alice})): &\quad \{N_A + 1\}_{K_{AB}} \\ Z(B) \to A: &\quad \{N_A + 1\}_{K_{AB}}. \end{aligned}$$

5.3.3 Public Key Distribution Protocol (Binding Attack)

Another (most likely unintentionally) flawed protocol is supposed to distribute public keys signed by the *certification authority* S:

$$\begin{aligned} A \to S: &\quad A,\, B,\, N_A \\ S \to A: &\quad S,\, \{S,\, A,\, N_A,\, K_B\}_{K_S^{-1}}. \end{aligned}$$

Unlike in other protocols, there are no secret messages send here, but the authority *signs* the message it sends to A, so if she trusts S, then she can be sure that K_B is indeed the public key of B, since only S could have generated a signature. However, *Athena* shows that there is a way to fool A in accepting a wrong public key in just 5 steps (generating 15 states). The attacker exploits the fact that the second message does not mention whose public key is being sent. So, he tempers with the first message replacing B with Z (his own name), and then lets the server reply to A with the wrong information, namely, his own public key. Now he can read everything that A will send to B, while B would not be able to read A's messages.

$$\begin{aligned} A \to Z(S): &\quad A,\, B,\, N_A \\ Z(A) \to S: &\quad A,\, Z,\, N_A \\ S \to A: &\quad S,\, \{S,\, A,\, N_A,\, K_Z\}_{K_S^{-1}}. \end{aligned}$$

The fixed version of this protocol (due to Hwang and Chen [HC95]) adds the B's identity to the last message inside the signature, and *Athena* proves it correct in just 4 steps.

5.3.4 ISO Symmetric Key Two-Pass Mutual Authentication

$$\begin{aligned} A \to B &: \; \{N_A, B\}_{K_{ab}} \\ B \to A &: \; \{N_B, A\}_{K_{ab}}. \end{aligned}$$

This protocol is taken from [CJ97] and simplified — the application-specific text fields are removed, since they are unessential to its security properties, but give *Athena* hard time. It suffers from so many flaws, which *Athena* finds immediately in a handful of steps, that it does not even deserve a thorough analysis, in our opinion. To mention a few, the nonces are never used and there is no way to check their freshness, so it's susceptible to all kinds of replay attacks. Also, if the text fields are added to the messages (in cleartext or under the encryption), then some other attacks may be possible without even requiring one of the participants. If timestamps are used instead of nonces, the replay attack might only be valid for a short period of time, and since *Athena* does not support timestamps, we cannot verify this version of the protocol. The paper [CJ97] does not mention anything about the integrity of this protocol.

5.3.5 ISO Symmetric Key Three-Pass Mutual Authentication

$$\begin{aligned} B \to A &: \; N_B \\ A \to B &: \; \{N_A, N_B, B\}_{K_{AB}} \\ B \to A &: \; \{N_B, N_A\}_{K_{AB}}. \end{aligned}$$

This is another ISO protocol with some application-specific text fields removed. As it is, the protocol is proven secure by *Athena* in just 12 steps. This is a bit surprising, as the last message does not bear any identifications apart from the nonces, but no replay or an oracle attack was found. The protocol might, however, become insecure when the optional text fields are added, depending on their contents. The paper [CJ97] has no comments on its correctness.

5.3.6 Andrew Secure RPC Protocol

$$\begin{aligned} A \to B &: \; A, \{N_A\}_{K_{AB}} \\ B \to A &: \; \{N_A + 1, N_B\}_{K_{AB}} \\ A \to B &: \; \{N_B + 1\}_{K_{AB}} \\ B \to A &: \; \{K'_{ab}, N'_B\}_{K_{AB}}. \end{aligned}$$

The protocol looks good up until the last message, which does not have anything to guarantee its freshness. Therefore, the intruder can simply replay the last message at a later date and forces A to accept an old (and invalid for this session) key. *Athena* finds this flaw automatically in 47 steps and 58 states, taking total under 12 seconds. The reason for a mild "state explosion" in this example is that we need to simulate two consecutive runs of the protocol, one successful, and one with the last message replaced by the intruder. This is exactly the attack described by Clark and Jacob [CJ97].

5.3.7 Otway-Rees Protocol

$$
\begin{array}{lll}
(1) & A \to B: & M,\, A,\, B,\, \{N_A,\, M,\, A,\, B\}_{K_{AS}} \\
(2) & B \to S: & M,\, A,\, B,\, \{N_A,\, M,\, A,\, B\}_{K_{AS}},\, \{N_B,\, M,\, A,\, B\}_{K_{BS}} \\
(3) & S \to B: & M,\, \{N_A,\, K_{ab}\}_{K_{AS}},\, \{N_B,\, K_{ab}\}_{K_{BS}} \\
(4) & B \to A: & M,\, \{N_A,\, K_{ab}\}_{K_{AS}}.
\end{array}
$$

Here M is another nonce generated by A, and is a run identifier. Clark and Jacob [CJ97] give a type flaw for this protocol, when a triple (M, A, B) can be confused for a new session key K_{ab}. Since our implementation of *Athena* works only on well-typed messages, this flaw would be difficult to find (if possible at all). However, we were able to find some other interesting facts about this protocol not mentioned in the survey.

One thing that *Athena* immediately finds about this protocol (in 7 steps) is that there is no way to check what messages actually were relayed through B in steps (2) and (3), since the intruder may reroute the message parts intended for S and A and send any garbage through B. To distract *Athena* from this minor nuisance, we rewrote the protocol in such a way that the parts in question are sent directly to the recipients, and B does not even have to know they exist.

The next thing that *Athena* discovers is that the session key K_{ab} is sent to A at the end of the protocol, and B never verifies that A has received it. This is not exactly a flaw, but at least a good fact to know. To prevent this "attack" we included *some* complete run of the participant A with the same M in the protocol, which *Athena* proves to be the same A and forces it to receive the key.

This, however, was not the end of the story. If the server does not check that the session identifier M is always new, then the following replay/integrity attack becomes possible. The intruder records the parts of messages (1) and (2) that are intended for the server S, and before the server has a chance to reply, he replays those parts again. The server, therefore, generates two copies of message (3) with different session keys K_{ab} and K'_{ab}.

But since the parts intended for A and B are not bound to each other, the attacker takes A's message from one session and B's message from the other session, puts them together as if the server's reply, and drops the other parts. The result is, A and B have different session keys and cannot communicate with each other. This can be considered a version of a DoS attack, and *Athena* finds it automatically in 16 steps, generating 65 states. This attack was found by Thayer, Herzog and Guttman [THG98a] in 1998.

Requiring the server to check for uniqueness of M (and disregarding the type attacks) makes the protocol secure, as proven by *Athena* with a substantial manual guidance in 551 steps, which took us an hour or two to complete. The proof contains a lot of repeated patterns, and we believe that with better proof search heuristics it can also be automated, even if at the expense of a longer proof.

5.3.8 SSH Public Key Client Authentication Algorithm

This is a real authentication protocol used in SSH version 2 [YKS+98a, YKS+98b] that provides one-way authentication of the client to the server. It is supposed to be run over the encrypted *transport layer protocol* [YKS+98c] that provides secrecy and integrity of messages. We model the transport layer by passing messages encrypted with a shared symmetric key (which is what it actually does). The protocol consists of one single message sent by the client to the server:

$$C \to S: \quad \{\text{session\_id},\, C,\, K_C,\, Text,\, \{\text{session\_id},\, C,\, K_C,\, Text\}_{K_C^{-1}}\}_{K_{CS}}.$$

Here session_id is another shared secret between the client and the server provided (together with the shared key) by the transport layer, and $Text$ is the rest of the message that identifies the type of request, the authentication method, etc. All parts of $Text$ are well-typed and can be clearly distinguished from any other part of the message. The authentication is provided by signing the entire message contents (without the signature itself), which proves to the server that the client possesses his private key.

Due to the technical restrictions on the input language, we defined the session id to be the same as the shared key K_{CS}, which after all must also be a shared secret and unique for each session.

There were no surprises with this protocol, and *Athena* proved it correct in 29 steps.

5.3.9 Bluetooth Authentication Protocol

This authentication protocol can be used within a Bluetooth network of wireless devices [Blu01], and its goal is to provide a secure authentication mechanism for remote devices when they are about to access some priveleged resources. The algorithm relies on the security of the underlying key exchange algorithms, which, as in the case of SSH, *Athena* is not designed to handle, and therefore, they have to be assumed secure. Below we assume that the required symmetric *link key* is shared by the two principals, *Master* and *Slave*, and the key is not yet compromised. The goal of the protocol is for the *Slavie* to prove its identity to the *Master* (one way authentication). If the authentication is required in both directions, this protocol is played second time in the opposite direction. The protocol consists of two messages:

$$M \to S : N_M$$
$$S \to M : \{S,\, N_M\}_{K_{SM}}$$

Here K_{SM} is the shared link key, and N_M is a nonce generated by the *Master* as a challenge to the *Slave*. As it is, the protocol is proven secure by *Athena* in just 5 steps.

5.3.10 The Overall Experience

The experiments above took us about a week to complete. The reason it took this long is because the tool was developed and refined as we were doing the experiments. This was, in fact, the main purpose of the exercise, and more important than the actual verification of the protocols. We have found that using the interactive user interface of *SyMP*, the built-in proof management, and the debugging support made the development of *Athena* very efficient and convenient.

One of the most helpful features turned out to be the interactive nature of the prover. During the development stage, when the automatic proof search failed to find a known attack and (erroneously) proved the property, we could guide the prover manually towards the known attack and see where it fails to find it, thus locating a soundness bug. The opposite case has also happened several times when the proof failed with an invalid counterexample (unprovable subgoal). Such an ability to play with the proof dynamically as it is being constructed has helped us immensely in finding many errors and subtleties not only in the implementation itself, but even in our understanding of the underlying theory. For example, it was not immediately obvious that the strand's identities (`Self` parameters) must be constrained such that the intruder cannot share identities with the regular principals, even those in Δ (the conclusion part of the sequent). However, no more constraints should be placed on the identities. In particular, it is unsound to require that different roles must have different identities, since the same principal might want to play different roles, and this ability may be required to mount an attack on the protocol.

Symmetric key protocols that generate fresh session keys suggested that roles should be able to generate fresh values other than nonces. This resulted in a very useful extension to the input language. A soundness bug was then found which resulted from the fact that shared symmetric key operator `SymKey(A,B)` was not commutative. After it has been made commutative, a few more protocols exhibited attacks that before were proven "secure." One such protocol is the symmetric key Needham-Schroeder protocol (see Section 5.3.1).

Another extension to the input language was suggested by the Otway-Rees protocol (Section 5.3.7) after we have found the flaw that relies on the server not checking the uniqueness of the session identifiers. We have added `unique` parameters to the roles that must be checked for uniqueness but do not have to be freshly generated by the role. This allowed us to specify that the server must not accept the same session identifier twice, and the protocol was proven to be secure. Of course, the area of security protocol verification has known cases when a protocol proven to be secure was indeed flawed, so we would not claim that *Athena* does not have any other subtle soundness errors. However, all the known flaws (apart from the type flaws that it is not designed to catch) were found in all the protocols we have experimented so far, and therefore, we believe that it is most likely sound.

The entire process of debugging and refining the *Athena* module in *SyMP* (not counting the implementation before the tests) took us about two person-weeks.

5.4 Combining Presburger Arithmetic with a Bit-Vector Theory

Practically all hardware designs contain operations on *bit-vectors*, including concatenation, sub-vector extraction, and boolean element-wise operations. Often these bit-vectors are then treated as integers, and arithmetic operations are performed on these same data. While there are plethoras of theories and decision procedures both for bit-vectors [Möl98] and for Presburger arithmetic [BC96, Opp78, Pug91, WB00] — a decidable subset of integer arithmetic often considered in formal verification, — there was no systematic combination of those that would allow more automatic verification of hardware designs with arithmetic.

While investigating the state of the art in both bit-vector and Presburger arithmetic decision procedures, we have found that there are many different approaches that, as it often happens, perform well in some cases and poorly in others, and an efficient and robust decision procedure could potentially be obtained by combining many existing techniques and picking one that performs the best for a given problem. Specifically, conjunctions of quantifier-free Presburger formulas can be solved most efficiently by Integer Linear Programming techniques based on the simplex method. Even if the formula is not a single conjunction, it can be converted to the Disjunctive Normal Form (DNF) and each disjunct solved separately. When DNF can be constructed, the *Omega-test* [Pug91] can be applied as an alternative. The DNF construction, however, may lead to exponential explosion of the formula size, and in some cases direct automata-based techniques can be more efficient, since they do not require any rewriting of a propositional formula. Such techniques have become reasonably efficient due to the use of Binary Decision Diagrams [Bry86] (BDDs) and similar structures, but are still significantly slower than simplex-based methods in general. As an alternative, the DNF for the formula may be constructed incrementally, and if the formula is satisfiable, there is a chance that a satisfying assignment will be found for one of the first few disjuncts.

When a formula contains quantifiers, simplex-based methods are no longer applicable. Automata-based techniques are available to handle quantifiers, but their complexity is not even known to be elementary in the worst case, and in practice is rather high. A feasible alternative is to translate such a formula syntactically to a quantifier-free formula using the triple-exponential algorithm by Oppen [Opp78]. The resulting formula often contains a lot of redundancies and must be heavily simplified. This, in turn, requires the development of many mostly *ad hoc* rewrite rules that require testing and tuning on practical examples.

Additionally, when a problem involves both bit-vector and arithmetic operations, only some extended automata-based techniques can be applied directly. Therefore, it is desirable to avoid the high complexity of these techniques by rewriting the problem in an equivalent form that makes the automata-based techniques more robust, or may even allow complete separation of the domains, so that more efficient specialized decision procedures can be used.

Since the problem in general contains so many parameters, and it is not immediately clear what heuristics are the best in practice, a prototype tool was needed to evaluate different approaches. The prototype must be relatively easy to implement, easy to reprogram with different strategies for choosing particular decision procedures or rewrite rules, and easy to add new external decision procedures and rewrite rules.

It turns out that our *SyMP* framework can be used to accomplish all of these goals. The external decision procedures and rewrite rules are implemented as inference rules, similar to the model checking-related rules in the default proof system. The strategies can be implemented on two levels: as general strategies provided by the *SyMP* proof manager, and as proof system's tactics. The former are extremely flexible, as the user writes the strategy at the interactive prover prompt. He can observe its performance, modify the stratergy, and try the new version immediately if the original one fails. However, the strategies are mostly limited to trying different rules until one applies, repeating the same sub-strategy, and running simple tests on the sequent for picking the next step. Tactics, on the other hand, are implemented in the proof system itself and have direct access to the sequent structure. Therefore, they can be much more intelligent and efficient in finding a proof than strategies. However, they are not as flexible as strategies, since they require recompilation of the proof system code for any serious modification.

Adding new decision procedures and rewrite rules is as simple (or as hard) as adding new inference rules to the proof system, and *SyMP* is designed to make this process as simple and modular as possible. On top of all, *SyMP* prover API encourages a well-typed input language, and we have chosen a simple but quite expressive First-Order language that includes Presburger arithmetic and Verilog bit-vector operations in a Verilog-like syntax.

At the moment of this writing, the high-level design of the proof system is complete, but the actual implementation is still under development. Building the basic infrastructure for this new proof system (the parser and typechecker for the input language, sequent and inference rule representations and required API to the *SyMP* proof manager) took us less than a week. Implementation of propositional and simple quantifier manipulation rules was nearly trivial and took another day or two just because of the number of rules and continued adjustments and debugging of the typechecker. The rules interfacing to the external decision procedures are being implemented at the time of this writing, and are expected to be the most demanding part of the process taking, possibly, two or three more weeks. This includes the implementation of our theory combining bit-vectors with Presburger arithmetic. Once this basic infrastructure is in place, we can begin our experiments, and further development will be example-guided.

5.5 *Reedpipe/CProver*: Verification of Embedded Software

This proof system has at least two important features that distinguish it from all the other proof systems described above. First, it is designed and implemented almost exclusively by Daniel Kröning, independently of the main *SyMP* developer. Second, the core of the proof system called *CProver* is implemented in C++, and only a small inteface layer (called *Reedpipe*) is in SML/NJ, the main implementation language of *SyMP*. This demonstrates that *SyMP* can indeed be used by others as a programmer's kit to build new theorem provers, and that it is not limited to any particular programming language.

The proof system itself is based on Hoare triples, and the logic includes constructs found in programming languages like C, so that the translation of the input language to logical formulas is rather straightforward. Besides the specialized inference rules targeted to the problem domain (which is very much in the *SyMP* spirit), the proof system is also based on the language-independent internal representation, and the actual input language and the printed syntax of the sequent can be dynamically changed by the user even while proving a theorem at the prover prompt. At the time of this writing it supported ANSI-C, *PVS*, and VHDL.

This system was inspired by Kröning's experience in verification of the VAMP microprocessor in *PVS* [Krö01, Krö99]. Althought quite successful overall, the verification process was at times overly complicated when it did not have to be. For example, some large finite-state subproblems that could be relatively easily proven with model checking were hard to prove in *PVS*. Certain types of abstraction techniques were mathematically elegant and simple, but when encoded in higher-order logic of *PVS* may become cumbersome and tedious to deal with. But most importantly, *PVS* lacked some efficient decision procedures that could help greatly in the verification of such kind of examples.

CProver is designed to be rich with various powerful decision procedures (bit-vectors, Presburger arithmetic, SAT solvers, simplex, model checking and other BDD-based procedures, etc.). It has several industry-standard input languages and a versatile internal representation adequate for embedded software and some hardware languages.

To date, the proof system is still under development, and significant examples are still to be tried, but this prover module already looks very promising from the smaller examples that have been verified.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

We have presented our work which consists of two most important contributions. First, we have shown that model checking can be efficiently combined with theorem proving in a way that sacrifices neither efficiency of the former, nor the expressiveness of the latter. We call this combination *model proving*. One of the important lessons that we learned is that such a fusion must be specific to the problem domain, and not too general, otherwise the efficiency is very hard to preserve. To address this issue, we have developed a framework for generalizing theorem proving (as it is usually accepted in the formal verification community) to arbitrary but narrow enough problem domains. This framework, along with its implementation in our tool *SyMP*, comprises our second major contribution in this work.

Although there has been much work in combining model checking and theorem proving in formal verification, we have not seen any results providing a general enough theory unifying both techniques *without loss of expressiveness and efficiency*. The latter is a very important aspect, since, theoretically, anything can be expressed in the Higher-Order Logic, but very often in practice working with such an encoding is virtually impossible. For instance, the simplest and most widely used method of adding model checking to theorem proving framework is to add a model checker as an external decision procedure in a special inference rule. The model is extracted from the sequent with the help of some heuristics or by matching certain formula templates for which the translation is easy. Many HOL-based theorem provers use this approach: *PVS*, *ACL2*, *HOL*, etc. This approach suffers from two drawbacks: first, the original structure of the model is lost, and often cannot be recovered and used to simplify the verification. Second, very often model checking is used only as a direct decision procedure for finite-state models to prove the current subgoal, and a plethora of reduction techniques developed in model checking is simply omitted in the hope that they can be done with the existing theorem proving techniques.

Another theorem prover, *STeP*, uses a similar idea in spirit, but more sophisticated in detail. Since *STeP* uses first-order LTL as a logic, the original model can be encoded in this logic and later extracted rather precisely in the model checking inference rule. Besides, other transformations that are usually attributed to model checking can be done efficiently (e.g. abstraction). However, *STeP* is highly specialized to software verification, its proof system is tightly integrated into the core of the tool, and therefore, it is not as general and flexible as our framework.

On the model checking side, the situation is almost reversed, but with the same overall effect. Whenever theorem proving techniques are added to a model checker, they usually stand separately, and often can only be activated at a "preprocessing" stage before the main model checking run. In addition, the theorem proving add-ons almost always consist of a fixed set of rules applied completely automatically, and therefore, not as adaptive to the problem domain or a particular problem as an interactive one.

One of the most often used reductions in model checking that may require some theorem proving capabilities is abstraction. In fact, it is used in the model checking context so often that it has become a standard practice and an integral part of the model checking process. Many tools apply abstraction to reduce the model to a manageable size and then run a model checking engine directly on the abstracted model. The theorem proving component here is to derive that if a property holds in the abstracted model, then it must hold in the original model. As it is, this "inference rule" is implemented only implicitly by checking the applicability conditions and abstracting the model. A few model checkers implement more theorem proving rules: assume-guarantee reasoning (*Mocha*), induction, case split, and use of lemmas (*SMV*), and so on. However, all these tools still have the set of rules hard-wired into the structure of the implementation. This makes them very specialized to a particular problem domain where they can be extremely efficient, but they perform poorly outside of it.

In our framework, the inference rules are always defined explicitly, and the user has a control in the amount of automation while applying these rules. In particular, if the automated verification goes terribly wrong, there is an option to guide it manually. Also, an explicit notion of an inference rule provides better modularity, and new reductions and transformations can be added easily and systematically.

Another practical advantage of using this framework is the appropriate level of abstraction of the user interface with the tool. When the proof system is designed specifically for the problem domain, the actual proofs in the tool correspond very closely to their mathematical description. For example, the proofs for the IBM cache coherence protocol described in Section 5.1 correspond almost exactly, both in the level of abstraction and in the sequence of steps, to the proofs in the default proof system of our tool *SyMP*. Given the original model description, one can easily repeat the proofs in *SyMP* while reading their description with little extra work. Moreover, the proof of the coherence property can be done using just theorem proving techniques or using more model checking-oriented methodology. We have essentially repeated *PVS* proofs step by step in *SyMP* without any difficulties, and then proved the same property by agressive use of abstraction and model checking closely following the *SMV* proof discovered by Ken McMillan.

Our second contribution is closely related to the ways the research is carried out in formal verification. This research area is very experimental, and new methodologies must be implemented and tested on real examples. Whenever a researcher comes up with a new methodology, he or she always faces a design decision of how to implement it. Development of formal verification tools is often an arduous process, and one of the big reasons for developing very general tools is an attempt to capture as many aspects of formal verification as possible in one single implementation. The experiments then can be carried out in such a tool as soon as one provides an encoding of his or her verification problem in the supported specification language. The other extreme that is often taken is to develop a new highly specialized tool for each problem domain or even each methodology, which results in a very efficient but not as flexible implementation.

Our experience with several tools both from model checking and theorem proving indicates that only sufficiently specialized ones can be very efficient in practice, and very general solutions quite often fail to exploit the shortcuts and powerful reductions particular to certain classes of examples, because what works great in one problem domain may be making things worse or even be unsound in another. But instead of writing yet another highly specialized tool or using a very general one, we decided to choose the middle-ground. We have designed and implemented our tool *SyMP* as a general framework for creating *specialized verification modules*. Therefore, although the tool itself is very general, we solve the problem of efficiency by adding modules specialized to different problem domains. That is, unlike in the other tools, there can be many different specializations in *SyMP* at the same time, which makes our framework and the tool versatile.

Despite the differences in techniques and representation, there are still quite a bit of common features in the process of formal verification that can be used in virtually any problem domain, and the most important ones are the proof management mechanism and the interactive component of the user interface. This is not surprising, since whatever we are proving, we would like our proofs to be complete; and almost any proof can be formalized in a suitable proof system. Therefore, a tool that serves as a basis for specialized proof systems and provides this common functionality can significantly reduce the implementation time of new verification methodologies, and this comprises one of our major contributions in this thesis. To date, we have not yet seen a tool or a library providing similar functionality.

The idea of a generic tool for implementing custom proof systems is not new, however. *Logical Framework* has been developed in the area of *Automated Deduction*, and is one of our most closely matching competitors. However, most of the implementations of the logical framework require that a custom proof system be expressed in a fixed input language of the tool, and although it often provides powerful automated proof search for each new proof system, it is again only efficient for the problem domains that can be adequately expressed in the provided language. In particular, none of the proof systems implemented so far in *SyMP* can be efficiently encoded into the Logical Framework.

Customized proof systems in *SyMP* can be used not only for theorem proving-based methodologies, but also as a way to house several existing tools under one roof. This can be done at different levels: within the same proof system, and across different proof systems. For instance, *SyMP* already uses the CMU version of *SMV* as a decision procedure in the default proof system, and at the same time another proof system is being developed that consists entirely of an interface to an external prover written in C++. This type of combination provides the common user interface to different applications for different problem domains, but it does not allow these tools to communicate with each other. Another possibility is to use external tools as decision procedures or processing filters in different inference rules of the *same* proof system. This forces the output of the tools to be translated into the common sequent representation (which is only fixed for a particular proof system, and can be different in the others), and passed to another tool.

Similar idea has been proposed at SRI several years ago as a *SAL* project. One of the central goals was to develop a universal language that can be used as an intermediate representation in virtually any problem domain. External tools would be communicating among each other through this language, and adding a new tool would only require writing translators to and from *SAL* for that tool. As in any other general approach, the bottleneck of *SAL* is the intermediate language itself. In our opinion, it is impossible to come up with a universal representation that would work efficiently for every problem domain. Therefore, *SAL* will most likely be successful only for some problems that it can express elegantly and adequately.

In *SyMP*, we deliberately did not commit ourserves to any particular specification language or intermediate representation. This, of course, adds some work for the developers of new proof systems, but then each proof system can have its own common representation most suitable to the problem domain, and at that point the situation is similar to the original idea of *SAL*: each new external tool only needs a small translator to and from the common sequent representation, but this time the sequent is optimized to the problem domain, and overall, we hope that *SyMP* can bridge more tools and more efficiently than *SAL* and similar projects.

6.2 Future Work

There is still a lot of work to be done in this project, most of it is related to the development of our tool *SyMP*. First of all, the default proof system is far from being complete. The idea of this proof system is to provide a very general framework that includes model checking and theorem proving together. Ideally, we would like to see most of the existing theorem proving rules and model checking reduction techniques implemented in it.

The reason we want to be very general in this case is to be able to provide a quick prototyping environment for new verification methodologies. In research, at the time of trying out a new idea, the efficiency usually is not a big issue, and researchers often prefer a quick feedback, even at the expense of tedious manual guidance and not very natural encoding of the problem. This is what the default proof system should provide: a research-oriented prototyping environment for methodologies that require both model checking and theorem proving techniques. One would be able to use virtually any existing reduction technique and the full deductive power of a theorem prover to run through several examples, and then decide which reductions were crutial to their methodology, and what was the bottleneck. Later, when the new methodology is thoroughly tested, another specialized proof system can be developed that carries out the verification process much faster and more automatically for that specific class of problems.

Adding new proof systems to our tool is another dimension of its growth. *SyMP* has already proven to be a very convenient basis for the two proof systems developed in it (the default one and *Athena*), and there are three more projects going on at the time of writing that will add a reimplementation of *Analytica,* a new prover module *Reedpipe/CProver* specialized in embedded software written in C, and a prototype of a decision procedure for a combination of Presburger arithmetic and a Verilog-like bitvector theory.

In the future, more modules will be added to cover other specific areas of formal verification. For instance, there are some thoughts about adding a module for software verification. It is quite possible that in a few years *SyMP* will become a powerful tool capable of efficiently solving many formal verification problems in a wide range of problem domains.

Appendix A

Soundness of the Default Proof System

In this section, we argue about the soundness of the proof system for combining model checking and theorem proving described in Chapter 3. Formally, we need to proof the soundness theorem stated in that Chapter:

Theorem. ***(3.5.1, Soundness)*** *Given a model M, if $M;\, \Sigma;\, \Gamma \vdash \Delta$ is derivable in the proof system, then*

$$M|_\Sigma \models \bigwedge \Gamma \rightarrow \bigvee \Delta.$$

However, to make the argument more precise, we define a *restriction operator* on a model M more rigorously.

Definition A.1. Model $M = (S, \rightarrow, I)$ restricted to a formula ϕ is the model $M|_\phi = (S', \rightarrow', I')$ is derived from M as follows:

- $S' = S \cap [\![\phi]\!]$
- $I' = I \cap S'$
- $\rightarrow' = \rightarrow \cap S' \times S'$.

We also write $M|_\Sigma$ for a set of formulas Σ to mean $M|_{\bigwedge \Sigma}$.

Proof. *(of Theorem 3.5.1)* By induction over the derivation of $M;\, \Sigma;\, \Gamma \vdash \Delta$. The inductive step involves proving soundness of each inference rule in the proof system, and for each rule of the form

$$\frac{P_1 \quad \cdots \quad P_n}{C}$$

we have to show that if Theorem 3.5.1 holds for each of the premisses P_i, then it must hold for the conclusion C. In other words, if $M_i|_{\Sigma_i} \models \bigwedge \Gamma_i \rightarrow \bigvee \Delta_i$, for all $i = 1 \ldots n$, then $M_C|_{\Sigma_C} \models \bigwedge \Gamma_C \rightarrow \Delta_C$.

- The soundness of the Init and Init_I rules is obvious.

- The soundness of the MC rule (application of the model checking decision procedure) is also straightforward, since this rule checks whether the sequent holds exactly as specified in Theorem 3.5.1.

- All the propositional rules ($\neg_R$, $\neg_L$, $\wedge_L$, $\wedge_R$, etc., including the Cut rule) are straightforward, since the soundness is stated by constructing a single formula using propositional connectives, and simple recombination of subformulas does not change the final formula in the soundness statement.

- The Cut_I rule is a bit more complicated, but still pretty obvious. The second premiss ($M;\ \Sigma;\ \Gamma \vdash \Delta,\ \mathbf{AG}\,\phi$) proves that ϕ is an invariant of M, and therefore, it does not make the model stronger to add ϕ to the set of invariants Σ.

- Weakening and Strengthening rules (w_R and s_L) follow from the observation that if ψ is true in a model, then $\psi \vee \phi$ is also true for any ϕ. This observation applies directly to w_R. For s_L, rewriting the top-level implication in the soundness formula as a disjunction gives $\neg(\bigwedge \Gamma) \vee \bigvee \Delta$, and $\neg(\bigwedge(\Gamma,\ \phi)) \equiv \neg(\bigwedge \Gamma) \vee \neg\phi$, which again reduces to the above observation.

  Other types of weakening rules: w_I holds simply because ϕ is enforced by Σ to hold everywhere in the model. Therefore, it also holds in every initial state, and adding it to Γ does not change the assumption part of the implication in the soundness formula, semantically. The rule s_{Init} is sound because from the premisses we know that the sequent holds for a larger set of initial states; therefore, it also holds for any subset of the initial states taken as a new set of the initial states of M.

- The soundness of the monotonicity rule Mono follows from the definition of monotonicity for temporal logic formulas: if $[\![\phi_1]\!] \subseteq [\![\phi_2]\!]$, then $[\![\psi^+(\phi_1)]\!] \subseteq [\![\psi^+(\phi_2)]\!]$. Notice, that the Mono rule requires Γ and Δ to contain exactly one formula both in the premisses and the conclusion, and the premiss is proven for all the reachable states of M. Therefore, for any reachable state $s \in S_M$ we know that $\phi_1 \rightarrow \phi_2$, which implies $[\![\phi_1]\!] \subseteq [\![\phi_2]\!]$, and hence, the conclusion also holds.

- The rule $\forall\mathrm{Case}_R$ is sound because ϕ is simply replaced by a semantically equivalent formula $\forall x.\, t = x \to \phi$.

- We skip the proof of soundness for the rules for modalities, since they essentially replace formulas with semantically equivalent ones based on well-known equivalences.

- Rules for quantifiers divide into two groups: "universal" ($\forall^a_R$, $\exists^a_L$) and "existential" ($\exists_R$, $\forall_L$). We only prove the soundness of $\forall^a_R$ and $\exists_R$ rules, the other two rules are dual and the argument is exactly the same.

  The existential rule $\exists_R$ relies on the tautology $\phi(t) \to \exists x.\, \phi(x)$. If $M \models \psi \vee \phi(t)$ for some term t, where $\psi = \bigwedge \Gamma \to \bigvee \Delta$, then, due to the tautology, we also have $M \models \psi \vee \exists x.\, \phi(x)$.

  To prove soundness of the universal rule $\forall^a_R$, recall that the semantics of a sequent with uninterpreted symbols (like a) is the following:

$$M;\, \Sigma;\, \Gamma \vdash \Delta,\, \phi(a) \text{ iff for all } d \in D_a :\ M|_{e[a \leftarrow d]};\, \Sigma;\, \Gamma \vdash \Delta,\, \phi(a),$$

  where e is the *environment* which assigns interpretation to free variables in formulas. That is, the sequent must hold for all the models that differ from M only in the interpretation of a (so, a is no longer an uninterpreted symbol in those models). Writing the soundness statement for the premiss explicitly, thus, gives us the following:

$$\text{for all } d \in D_a :\ M|_{e[a \leftarrow d]} \models \psi \vee \phi(a),$$

  where $\psi = \bigwedge \Gamma \to \bigvee \Delta$. This, in turn, is equivalent to:

$$\text{for all } d \in D_a :\ I \subseteq [\![\psi]\!]_M e[a \leftarrow d] \cup [\![\phi(a)]\!]_M e[a \leftarrow d].$$

  Since a does not occur anywhere but in ϕ, the meta-quantifier can be pushed down to $\phi(a)$:

$$I \subseteq [\![\psi]\!]_M e \cup \Big(\bigcap_{d \in D_a} [\![\phi(a)]\!]_M e[a \leftarrow d] \Big),$$

  which, according to the semantics, is exactly the same as

$$I \subseteq [\![\psi]\!]_M e \cup ([\![\forall x.\, \phi(x)]\!]_M e),$$

  and hence we have

$$M;\, \Sigma;\, \Gamma \vdash \Delta,\, \forall x.\, \phi(x).$$

- The induction rules Ind_R^a and wf_Ind_R^a are derived from the propositional and quantifier rules and the equivalences valid in the natural arithmetics:

$$\forall x.\, \phi(x) \equiv \phi(0) \wedge \forall i.\, \phi(i) \rightarrow \phi(i+1), \text{ and}$$

$$\forall x.\, \phi(x) \equiv \forall i.\, \forall j < i.\, \phi(j) \rightarrow \phi(i),$$

  where x, i, and j range over a well-founded domain w.r.t. the order "$<$" in the second equivalence.

- The rules for unfolding fixpoints ($\nu\mathrm{fix}_R$, $\mu\mathrm{fix}_R$, etc.) are sound because they replace fixpoint formulas with semantically equivalent ones.

- The rules for eliminating fixpoints, similarly to the quantifier rules, group into two categories: "invariant" rules (ν_R and μ_L) and "termination" rules (μ_R^α and ν_L^α). Again, we prove soundness of only one rule from each category, namely ν_R and μ_R^α. Both categories rely on Tarski's definitions of the fixpoints:

$$[\![\nu X.\, \phi^+(X)]\!]e = \bigcup\{S \mid S \subseteq [\![\phi^+(X)]\!]e[X \leftarrow S]\} \text{ and}$$

$$[\![\mu X.\, \phi^+(X)]\!]e = \bigcap\{S \mid [\![\phi^+(X)]\!]e[X \leftarrow S] \subseteq S\}.$$

  The conclusion of the rule ν_R requires the following to hold:

$$I' \subseteq [\![\nu X.\, \phi^+(X)]\!]e,$$

  where $I' = I \cap [\![\wedge\Gamma \wedge \neg(\bigvee \Delta)]\!]e$, and I is the set of initial states of M. According to the Tarski's definition of the greatest fixpoint, it is sufficient to find *some* S such that $I' \subseteq S$ and $S \subseteq [\![\phi^+(X)]\!]e[X \leftarrow S]$. Taking $S = [\![\psi]\!]e$, where ψ is the formula from the premisses of ν_R, we can show that S satisfies these two conditions. Applying induction hypothesis to the first premiss immediately gives us $I' \subseteq S$. The second premiss similarly yields the second condition, since the implication $\psi \rightarrow \phi^+(\psi)$ is proven for all the reachable states of M w.r.t. Σ.

  The conclusion of the rule μ_R^α requires the following to hold:

$$I' \subseteq [\![\mu X.\, \phi^+(X)]\!]e.$$

  According to the Tarski's definition of the least fixpoint, we have to show that $I' \subseteq S$ for *every* S such that $[\![\phi^+(X)]\!]e[X \leftarrow S] \subseteq S$ (the post-fixpoint condition). Since this statement involves an implicit universal quantifier over S, we can eliminate it the same way we did in the regular quantifier rules, namely, introduce a new *uninterpreted propositional constant* α, and show that $I' \subseteq [\![\alpha]\!]e$ holds for any α that satisfies the post-fixpoint condition, which can be expressed as $[\![\phi^+(\alpha) \to \alpha]\!]e$. Since this condition must be universally true in the model M, we require it to be an invariant of M. Putting it all together, we obtain the following sufficient condition:

$$I' \subseteq [\![\alpha]\!]_{M|_{\Sigma,\, \phi^+(\alpha)\to\alpha}} e,$$

which is exactly the semantics of the premiss of the μ_R^α rule.

- The Cone of Influence rule (COI) replaces the model with a semantically equivalent one, so the rule is trivially sound.

- The soundness of the abstraction rule (Abs) follows from the fact that existential abstraction preserves $\Box$-μ-calculus formulas (those that only contain $\Box$ modalities under even number of negations, and $\diamond$ modalities under odd number of negations). This is similar to the preservation of ACTL [Lon93].

$\Box$

The soundness of the proof system for the linear time variant of the μ-calculus can be proven similarly. The only difference will be in the rules for modalities and the abstraction rule. All the other rules are independent of the type of modalities used, and those parts of the proof of Theorem 3.5.1 can be reused.

Appendix B

Examples of Security Protocols in *Athena* Input Language

Most of the following protocols are taken from "A Survey of Authentication Protocol Literature: Version 1.0" by John Clark and Jeremy Jacob [CJ97].

Original Needham-Schröder public key authentication protocol. *Athena* automatically finds the Lowe's attack.

```
protocol NSOld =
  begin

    role Init [NA: Fresh Nonce; NB: Nonce;
               A: Self; B: Principal] =
      begin
        start: send {(NA, A)} (PK B)
        receive {(NA, NB)} (PK A)
        finish: send {NB} (PK B)
      end

    role Resp [NA: Nonce; NB: Fresh Nonce;
               A : Principal; B : Self] =
      begin
        start: receive {(NA, A)} (PK B)
        respond: send {(NA, NB)} (PK A)
        finish: receive {NB} (PK B)
      end
```

```
    predicate responded[NA: Nonce; NB: Nonce; A, B: Princi-
pal] =
      Resp[NA,NB,A,B]

    predicate initiated[NA: Nonce; NB: Nonce; A, B: Princi-
pal] =
      Init[NA,NB,A,B].finish

    theorem agreement[NA, NB: Nonce; A, B: Principal] =
      (initiated[NA,NB,A,B] -> Resp[NA,NB,A,B].respond)
      and (responded[NA,NB,A,B] -> initiated[NA, NB, A, B])

  end -- NSOld
```

Neehdam-Schröder public key authentication protocol with the Lowe's fix. Athena proves this protocol secure automatically.

```
protocol NSA =
  begin

    role Init [NA: Fresh Nonce; NB: Nonce;
               A: Self; B: Principal] =
      begin
        start: send {(NA, A)} (PK B)
        receive {(NA, NB, B)} (PK A)
        finish: send {NB} (PK B)
      end

    role Resp [NA: Nonce; NB: Fresh Nonce;
               A : Principal; B : Self] =
      begin
        start: receive {(NA, A)} (PK B)
        respond: send {(NA, NB, B)} (PK A)
        finish: receive {NB} (PK B)
      end

    predicate responded[NA: Nonce; NB: Nonce; A, B: Princi-
pal] =
       Resp[NA,NB,A,B]
    predicate initiated[NA: Nonce; NB: Nonce; A, B: Princi-
pal] =
```

```
     Init[NA,NB,A,B].finish

   theorem agreement[NA, NB: Nonce; A, B: Principal] =
     (initiated[NA,NB,A,B] -> Resp[NA,NB,A,B].respond)
     and (responded[NA,NB,A,B] -> initiated[NA, NB, A, B])

 end -- NSA
```

Symmetric key Needham-Schröder protocol. To our surprise, *Athena* automatically finds two new attacks previously not covered in the literature.

```
protocol NSPsymm =
  begin
    -- Initiator
    role Alice[A: Self; B, S: Principal; Kab: SymKey;
               Bmsg: Message;
               Na: FreshNonce; Nb: Nonce] =
      begin
        send (A,B,Na)
        receive { (Na, B, Kab, Bmsg) } SymKey(A, S)
        send Bmsg
        receive { Nb } Kab
        -- We cannot construct Nb-1, so we do
        -- another well-known operation on Nb.
        -- This should be equivalent.
        send { (Nb, Nb) } Kab
      end

    -- responder
    role Bob[A: Principal; B: Self; S: Principal;
             Kab: SymKey; Nb: FreshNonce] =
      begin
        receive { (Kab, A) } SymKey(B, S)
        responded: send { Nb } Kab
        finish: receive { (Nb, Nb) } Kab
        -- Disclose a key from the completed session
       -- This was supposed to help mount a known attack,
       -- but wasn't used as new attacks were found
        send Kab
      end

    role Server[A, B: Princi-
pal; S: Self;  Kab: fresh SymKey; Na: Nonce] =
```

```
        begin
          receive (A, B, Na)
          send {(Na, B, Kab, { (Kab, A) } SymKey(B, S))} SymKey(A, S)
        end

      -- We need the server in the assumption to make sure
      -- Bob knows who the server is.
      --
Otherwise the intruder can play the server's role for Bob
      -- and you know what happens...
      theorem BobSeesAlice[A, B, S: Principal; Na, Nb: Nonce;
                           Kab, Kab': SymKey] =
        (Bob[A,B,S,Kab,Nb].finish and Server[A, B, S, Kab', Na]
          -> Alice[A,B,S,Kab, {(Kab, A) } SymKey(B, S), Na,Nb])

      theorem AliceSeesBob[A, B, S: Princi-
pal; Na, Na', Nb: Nonce;
                           Kab, Kab': SymKey; M: Message] =
        (Alice[A,B,S,Kab, M, Na,Nb] and Server[A,B,S,Kab',Na']
          -> Bob[A,B,S,Kab,Nb].responded)

-- The freshness of the key cannot be guaranteed.
-- This theorem states that no two runs of Bob
-- (exactly the same principal, or other principals playing
-- the same role) can have the same session key,
-- which fails with a counterexample.

      theorem BobFresh[A, B, B2, S: Principal;
                       Na, Na2, Na', Nb, Nb2: Nonce;
                       Kab, Kab2, Kab': SymKey; M, M2: Mes-
sage] =
        (Alice[A,B,S,Kab, M, Na,Nb] and Server[A,B,S,Kab',Na']
         and Bob[A,B,S,Kab,Nb]
          -> not Bob[A,B2,S,Kab,Nb2].finish)

    end
```

A simple intentionally flawed protocol on page 26 of [CJ97].

```
protocol Parallel =
  begin
```

```
    role Alice[A: Self; B: Principal; Na: fresh nonce] =
      begin
        send {Na} SymKey(A,B)
        -- We cannot form Na+1, but we can (Na,Na),
        -- which must be equivalent.
        receive {(Na, Na)} SymKey(A,B)
      end

    role Bob[A: Principal; B: Self; Na: nonce] =
      begin
        receive {Na} SymKey(A,B)
        send {(Na, Na)} SymKey(A,B)
      end

    theorem correct[A,B: Principal; Na: nonce] =
      Alice[A,B,Na] -- and not Alice[A,A,Na]
      -> Bob[A,B,Na]
  end
```

Next two protocols are from the page 33 of Clark & Jacob [CJ97] (sec. 4.5: Binding Attacks).

```
protocol CA =
  begin
    role Client[C: Self; S, AS: Princi-
pal; Nc: fresh nonce; Ks: PubKey] =
      begin
        send (C, S, Nc)
        receive (AS, { (AS, C, Nc, Ks) } PVK AS)
      end

    role Authority[C, S: Principal; AS: Self; Nc: nonce] =
      begin
        receive (C, S, Nc)
        send (AS, { (AS, C, Nc, PK S) } PVK AS)
      end

    -- If C completes the protocol, the key it gets is (PK S).
    theorem correctKey[C, S, AS: Princi-
pal; Nc, Nc': nonce; Ks: PubKey] =
      Client[C, S, AS, Nc, Ks] and Authority[C, S, AS, Nc]
```

```
        -> Client[C, S, AS, Nc, PK S]

  end

protocol CAfixed =
  begin
    role Client[C: Self; S, AS: Principal;
                Nc: fresh nonce; Ks: PubKey] =
      begin
        send (C, S, Nc)
        receive (AS, { (AS, C, Nc, S, Ks) } PVK AS)
      end

    role Authority[C, S: Principal; AS: Self; Nc: nonce] =
      begin
        receive (C, S, Nc)
        send (AS, { (AS, C, Nc, S, PK S) } PVK AS)
      end

    -- If C completes the protocol, the key it gets is (PK S).
    theorem correctKey[C, S, AS: Principal;
                       Nc, Nc': nonce; Ks: PubKey] =
      Client[C, S, AS, Nc, Ks] and Authority[C, S, AS, Nc]
        -> Client[C, S, AS, Nc, PK S]

  end
```

Authentication protocol from the ISO 613 standard.

```
protocol ISO_613 =
  begin
-- role Alice[A: Self; B: Principal;
--            Na: fresh nonce; Nb: nonce;
--            T1, T2: new message; T3, T4: Message] =
--       begin
--      send (T2, { (Na, B, T1) } SymKey(A, B))
--      receive (T4, { (Nb, A, T3) } SymKey(A, B))
--       end

-- role Bob[A: Principal; B: Self;
--          Na: nonce; Nb: fresh nonce;
--          T1, T2: message; T3, T4: new Message] =
--       begin
```

```
--        receive (T2, { (Na, B, T1) } SymKey(A, B))
--        send (T4, { (Nb, A, T3) } SymKey(A, B))
--         end


-- Text fields confuse Athena:
-- "what if Alice sends Bob's response in T2?".
-- They are irrelevant to the protocol anyways,
-- so we remove them.
role Alice[A: Self; B: Principal; Na: fresh nonce; Nb: nonce;
           T1, T3: Message] =
  begin
    started: send { (Na, B, T1) } SymKey(A, B)
    receive { (Nb, A, T3) } SymKey(A, B)
  end

role Bob[A: Principal; B: Self; Na: nonce; Nb: fresh nonce;
         T1, T3: Message] =
  begin
    receive { (Na, B, T1) } SymKey(A, B)
    send { (Nb, A, T3) } SymKey(A, B)
  end

-- If each party is not talking to him/herself,
-- then there must be the other corresponding party
-- in the protocol.
theorem correct[A, B: principal; Na, Nb: nonce;
                T1, T3: Message] =
     (Alice[A, B, Na, Nb, T1, T3]
       and not Alice[A, A, Na, Nb, T1, T3]
    -> Bob[A, B, Na, Nb, T1, T3])
 and (Bob[A, B, Na, Nb, T1, T3] and not Bob[B, B, Na, Nb, T1, T3]
   -> Alice[A, B, Na, Nb, T1, T3].started)
end
```

Authentication protocol from the ISO 614 standard. Again, we eliminate the clear-text messages. In fact, here they are bad even under the encryption, since Athena cannot bound the encryption depth, and the most crucial pruning theorem doesn't apply.

```
protocol ISO_614 =
  begin
```

```
    role Alice[A: Self; B: Princi-
pal; Ra: fresh nonce; Rb: nonce] =
      begin
        receive Rb
        responded: send {(Ra, Rb, B)} SymKey(A,B)
        receive {(Rb, Ra)} SymKey(A,B)
      end

    role Bob[A: Princi-
pal; B: Self; Ra: nonce; Rb: fresh nonce] =
      begin
        send Rb
        receive {(Ra, Rb, B)} SymKey(A,B)
        send {(Rb, Ra)} SymKey(A,B)
      end

    theorem correct[A,B: Principal; Ra, Rb: nonce] =
      (Alice[A,B,Ra,Rb] -> Bob[A,B,Ra,Rb])
      and (Bob[A,B,Ra,Rb] -> Alice[A,B,Ra,Rb].responded)

  end

-- Instead of N+1 for a nonce N we construct (N,N).
-- It should be equivalent.
```

Andrew secure RPC protocol.

```
protocol AndrewRPC_616 =
  begin
    role Alice[A: Self; B: Principal; Na: fresh nonce;
               Nb, Nb': nonce; Kab': SymKey] =
      begin
        send (A, {Na} SymKey(A,B))
        receive { ((Na, Na), Nb) } SymKey(A,B)
        responded: send { (Nb, Nb) } SymKey(A,B)
        receive { (Kab', Nb') } SymKey(A,B)
      end

    role Bob[A: Principal; B: Self; Na: nonce;
             Nb, Nb': fresh nonce; Kab': fresh SymKey] =
      begin
        receive (A, {Na} SymKey(A,B))
        send { ((Na, Na), Nb) } SymKey(A,B)
```

```
        receive { (Nb, Nb) } SymKey(A,B)
        send { (Kab', Nb') } SymKey(A,B)
      end

    theorem correct[A,B: Principal; Na, Nb, Nb': nonce;
                    Kab': symkey] =
      (Alice[A,B,Na,Nb,Nb',Kab'] -> Bob[A,B,Na,Nb,Nb',Kab'])

  end
```

Otway-Rees protocol. Bob is supposed to forward the message from the server to Alice. But since he has no idea what he's resending, the intruder may funnel anything through him without any problem, and redirect the right part of message to Alice by some other route. So, we assume the server sends it to Alice directly, Bob never even receives it. Similarly the message from Alice to the server doesn't have to go through Bob.

Next point: when one party finishes, there is no guarantee that the other party actually received the session key, and we cannot prove anything useful. Therefore, we assume that we also have *some* strand of the other party which is complete and must have been from the same protocol run. This will force the strand to receive the key.

```
protocol OtwayRees_633 =
  begin
    role Alice[A: Self; B, S: Princi-
pal; M: fresh nonce; Na: fresh nonce; Kab: SymKey] =
      begin
        started: send (M, A, B, {(Na, M, A, B)} SymKey(A, S))
        receive (M, {(Na, Kab)} SymKey(A, S))
      end

    role Bob[A, S: Princi-
pal; B: Self; M: nonce; Nb: fresh nonce;
             Kab: SymKey] =
      begin
        receive (M, A, B)
        send (M, A, B, {(Nb, M, A, B)} SymKey(B, S))
        receive (M, {(Nb, Kab)} SymKey(B, S))
        send M
-- The other part of the message is left
-- as an exercise to the intruder
      end
```

```
    -- The attack exists when Server doesn't check for
    -- uniqueness of M.  If we require uniqueness, and
    -- assume that message types are inforced,
    -- then Athena proves the protocol secure.
    role Server[A, B: Principal; S: Self; M: unique nonce;
                Na, Nb: nonce; Kab: fresh symkey] =
      begin
        receive (M, A, B, {(Na, M, A, B)} SymKey(A, S),
                          {(Nb, M, A, B)} SymKey(B, S))
        send (M, {(Na, Kab)} SymKey(A,S),
                 {(Nb, Kab)} SymKey(B,S))
      end

    -- Although it is not necessary to find an attack,
    -- it's more illustrative to require
    -- that Alice or Bob are not trying to talk to themselves.
    theorem correct[A,B,S: principal;
                    M, M', Na, Nb, Na', Nb': nonce;
                    Kab, Kab', Kab": symkey] =
      Server[A,B,S,M',Na',Nb',Kab'] ->
        (Alice[A,B,S,M,Na,Kab]
         and not Alice[A,A,S,M,Na,Kab]
         and Bob[A,S,B,M,Nb', Kab"]
           -> Bob[A,S,B,M,Nb', Kab])
       and (Bob[A,S,B,M,Nb, Kab]
            and not Bob[B,S,B,M,Nb, Kab]
            and Alice[A,B,S,M,Na',Kab"]
              -> Alice[A,B,S,M,Na',Kab])
  end
```

The public key authentication method used by SSH This protocol is taken almost as it is from [YKS+98a], but some parts abstracted. M consists of some service information (message type identifier, service name, string *"publickey"* to identify the type of authentication, boolean `true`, and public key algorithm name), and can be clearly distinguished from any other message (i.e. it is well-typed), so we give it some atomic type not used in the protocol, namely '`Nonce`'.

```
protocol SSHpkauth =
```

```
begin
  role Client[C: self; S: Principal; M: nonce] =
    begin
      send {(symkey(C,S), C, PK C, M,
            {(symkey(C,S), C, PK C, M)} PVK C)} symkey(C,S)
    end

  role Server[C: principal; S: self; M: nonce] =
    begin
      receive {(symkey(C,S), C, PK C, M,
               {(symkey(C,S), C, PK C, M)} PVK C)} symkey(C,S)
    end

  theorem correct[C,S: principal; M: nonce] =
    Server[C,S,M] -> Client[C,S,M]

end
```

Bluetooth Authentication protocol from the original specification [Blu01].

```
protocol BluetoothAuth =
  begin
    -- One-way authentication
    role Master[A: Self; B: principal; Na: fresh nonce] =
      begin
        -- First, generate new link key one direction
        start: send Na
        receive {(B, Na)} SymKey(A,B)
      end

    role Slave[A: principal; B: self ; Na: nonce] =
      begin
        receive Na
        send {(B, Na)} SymKey(A,B)
      end

    -- One-way authentication is secure.  Proven in 5 steps.
    theorem correct[A,B: principal; Na: nonce] =
      Master[A,B,Na] -> Slave[A,B,Na]

    -- Two-way authentication is implemented as two runs
    -- of one-way authentication.
  end
```

Index

Bibliography

[AH96] R. Alur and T. A. Henzinger. Reactive modules. In *the proceedings of the 11th annual IEEE Symposium on Logic in Computer Science (LICS '96)*, 1996.

[AHM+98] R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Tasiran. MOCHA: modularity in model checking. In Hu and Vardi [HV98], pages 521–525.

[AJS98] Mark Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Combining theorem proving and trajectory evaluation in an industrial environment. In *Design Automation Conference*, pages 538–541, 1998.

[AL93] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.

[ASW94] Andersen, Stirling, and Winskel. A compositional proof system for the modal μ-calculus. In *LICS: IEEE Symposium on Logic in Computer Science*, 1994.

[BBC+99] N. Bjørner, A. Browne, M. Colon, B. Finkbeiner, Z. Manna, H. Sipma, and T. Uribe. Verifying temporal properties of reactive systems: A STeP tutorial. *Formal Methods in System Design*, 1999.

[BC96] Alexandre Boudet and Hubert Comon. Diophantine equations, Presburger arithmetic and finite automata. In H. Kirchner, editor, *Colloquium on Trees in Algebra and Programming*, volume 1059 of *Lecture Notes in Computer Science*, pages 30–43. Springer Verlag, 1996.

[BCC97] Sergey Berezin, Sergio Campos, and Edmund M. Clarke. Compositional reasoning in model checking. In *COMPOS'97*, volume 1536 of *LNCS*. Springer-Verlag, September 1997.

[BD94] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In D. L. Dill, editor, *Computer Aided Verification (CAV'94)*, volume 818 of *LNCS*. Springer-Verlag, 1994.

[BDL96] Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods In Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201. Springer-Verlag, November 1996. Palo Alto, California, November 6–8.

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

[BI01] Christoph Berg and Christian Jacobi II. Formal verification of the VAMP floating point unit. In *Conference on Correct Hardware Design and Verification Methods*, pages 325–339, 2001.

[BLO98] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Invest: A tool for the verification of invariants. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-Aided Verification (CAV'98)*, number 1427 in Lecture Notes in Computer Science, pages 505–510, Vancouver, Canada, June 1998. Springer-Verlag.

[Blu01] *Specification of the Bluetooth System, version 1.1*. Bluetooth SIG Inc., February 2001.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[CJ95] E. M. Clarke and S. Jha. Symmetry and induction in model checking. In J. Van Leeuwen, editor, *Computer science today: recent trends and developments*, number 1000 in Lecture Notes in Computer Science. Springer-Verlag, 1995.

[CJ97] J. Clark and J. Jacob. A survey of authentication protocol literature: Version 1.0, November 1997.

[CZ92] Edmund Clarke and Xudong Zhao. Analytica — A theorem prover for mathematica. In *Automated Deduction-CADE-II*, pages 761–763, 11th International Conference on Automated Deduction, Saratoga Springs, New York, June 15-18 1992.

[CZ93] Edmund Clarke and Xudong Zhao. Analytica: A theorem prover for *Mathematica*. *The Mathematica Journal*, 1993.

[FHG98] F. Javier Thayer Fabrega, Jonathan C. Herzog, and Joshua D. Guttman. Honest ideals on strand spaces. In *In Proceedings of 1998 Computer Security Foundations Workshop*, June 1998.

[Gre98] David Greve. Symbolic simulation of the JEM1 microprocessor. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD '98)*, volume 1522 of *Lecture Notes in Computer Science*, pages 321–333, Palo Alto, CA, November 1998. Springer-Verlag.

[GS96] Susanne Graf and Hassen Saïdi. Verifying invariants using theorem proving. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 196–207, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[HC95] Tzonelih Hwang and Yung-Hsiang Chen. On the security of SPLICE/AS: The authentication system in WIDE Internet. *Information Processing Letters*, 53:97–101, 1995.

[HM85] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the Association for Computing Machinery*, 32(1):137–161, mar 1985.

[HSG98] R. Hosabettu, M. Srivas, and G. Gopalakrishnan. Decomposing the proof of correctness of pipelined microprocessors. In Hu and Vardi [HV98], pages 122–134.

[HV98] Alan J. Hu and Moshe Y. Vardi, editors. *Computer Aided Verification (CAV'98)*, volume 1427 of *LNCS*. Springer-Verlag, June 1998.

[IEE94] IEEE Computer Society. *IEEE Standard for Futurebus+—Logical Protocol Specification*, 1994. IEEE Standard 896.1, 1994 Edition.

[KM00] Matt Kaufmann and J. Strother Moore. ACL2 (documentation, version 2.5). `http://www.cs.utexas.edu /users /moore /publications /acl2-book.ps.gz`, 2000.

[Koz83] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, December 1983.

[Krö99] Daniel Kröning. Design and evaluation of a RISC processor with a Tomasulo scheduler. Master's thesis, University of Saarland, Computer Science Department, Germany, 1999.

[Krö01] Daniel Kröning. *Pipelined Microprocessors*. PhD thesis, University of Saarland, Computer Science Department, Germany, 2001.

[Lon93] D. E. Long. *Model Checking, Abstraction, and Compositional Verification*. PhD thesis, Carnegie Mellon University, 1993.

[Low95] Gavin Lowe. An Attack on the Needham-Schroeder Public Key Authentication Protocol. *Information Processing Letters*, 56(3):131–136, 1995.

[McM93] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.

[McM98] K. L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In Hu and Vardi [HV98].

[McM99] K. L. McMillan. Circular compositional reasoning about liveness. Technical report, Cadence Berkeley Labs, Cadence Design Systems, 1999.

[MCon81] J. Misra, K. Chandy, and o networks. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, SE-7(4):417–426, 1981.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535, 2001.

[Möl98] M. Oliver Möller. Solving bit-vector equations - a decision procedure for hardware verification, 1998. Diploma Thesis, available at `http://www.informatik.uni-ulm.de/ki/Bitvector/`.

[NS78] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[Opp78] Derek C. Oppen. A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic. *Computer and System Sciences*, 16(3):323–332, June 1978.

[Par81] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *5th GI-Conference on Theorical Computer Science*, volume 104 of *LNCS*, pages 561–572. Springer Verlag, mar 1981.

[Pfe94] Frank Pfenning. Logical frameworks. `http:// www.cs.cmu.edu/ ~fp/ lfs.html` on the World-Wide Web, October 1994.

[Pfe99a] Frank Pfenning. Logical and meta-logical frameworks. In G. Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, page 206, Paris, France, September 1999. Springer-Verlag LNCS. Abstract of invited talk.

[Pfe99b] Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science Publishers, 1999. In preparation.

[Pnu85] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K.R. Apt, editor, *Logics and Models of Concurrent Systems, sub-series F: Computer and System Science*, pages 123–144. Springer-Verlag, 1985.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[Pug91] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.

[SBP01] Dawn Song, Sergey Berezin, and Adrian Perrig. Athena: a novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9:47–74, 2001.

[SG97] Hassen Saïdi and Susanne Graf. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer-Aided Verification, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 1997. Springer-Verlag.

[Son99] Dawn Song. Athena: An automatic checker for security protocol analysis. In *Proceedings of the 12th Computer Science Foundation Workshop*, 1999.

[SOR93] N. Shankar, S. Owre, and J. M. Rushby. *PVS Tutorial*. Computer Science Laboratory, SRI International, Menlo Park, CA, 1993. Also appears in Tutorial Notes, *Formal Methods Europe'93: Industrial-Strength Formal Methods*, pages 357–406, Odense, Denmark, April 1993.

[SS90] G. Stalmarck and M. Saflund. Modelling and verifying systems and software in propositional logic. In *Proceedings of SAFECOMP'90*. IFAC, Pergamon Press, 1990.

[SS96] J. P. M. Silva and K. A. Sakallah. GRASP – A new search algorithm for satisfiability. In *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design*, pages 220–227, 11 1996.

[Tar55] A. Tarski. A lattice-theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[THG98a] Thayer, Herzog, and Guttman. Honest ideals on strand spaces. In *PCSFW: Proceedings of The 11th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1998.

[THG98b] F.Javier Thayer, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Why is a security protocol correct? In *Proceedings of 1998 IEEE Symposium on Security and Privacy*, 1998.

[THG99] F. Javier Thayer Fábrega, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: proving security protocols correct. *Journal of Computer Security*, 7:191 (40 pages), January 1999.

[Wal95] Igor Walukiewicz. Completeness of Kozen's axiomatisation of the propositional μ-calculus. *LICS'95*, pages 14–24, 1995.

[WB00] Pierre Wolper and Bernard Boigelot. On the construction of automata from linear arithmetic constraints. In *Proc. 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, pages 1–19, Berlin, March 2000. Springer-Verlag.

[YKS$^+$98a] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen. SSH authentication protocol. Internet Draft at `http://www.ssh.com`, August 1998.

[YKS$^+$98b] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen. SSH protocol architecture. Internet Draft at `http://www.ssh.com`, August 1998.

[YKS$^+$98c] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen. SSH transport layer protocol. Internet Draft at `http://www.ssh.com`, August 1998.

[ZS94] H. Zhang and M. Stickel. Implementing Davis-Putnam's method. Technical report, The University of Iowa, 1994.