Formal Verification of a Partial-Order Reduction Technique for Model Checking

Ching-Tsun Chou*
Doron Peled†

ABSTRACT Mechanical theorem proving and model checking are the two main methods of formal verification, each with its own strengths and weaknesses. While mechanical theorem proving is more general, it requires intensive human guidance. model checking is automatic, but is applicable to a more restricted class of problems. It is appealing to combine these two methods in order to take advantage of their different strengths. Prior research in this direction has focused on how to decompose a verification problem into parts each of which is manageable by one of the two methods. In this paper we explore another possibility: we use mechanical theorem proving to formally verify a *meta-theory* of model checking. As a case study, we use the mechanical theorem prover HOL to verify the correctness of a *partial-order reduction* technique for cutting down the amount of state search performed by model checkers. We choose this example for two reasons. First, this reduction technique has been implemented in the protocol analysis tool SPIN to significantly speed up the analysis of many practical protocols; hence its correctness has important practical consequences. Second, the correctness arguments involve nontrivial mathematics, the formalization of which we hope will become the basis of a formal meta-theory of many model checking algorithms and techniques. Our formalization led to a nontrivial generalization of the original informal theory. We also discuss the lessons, both encouraging and discouraging, learned from this exercise.

1 Introduction

Mechanical theorem proving and model checking are the two main methods of formal verification, each with its own strengths and weaknesses. Mechanical theorem proving is more general, but requires intensive human guidance. In practice, only experienced experts can use mechanical theorem

*Fujitsu Laboratories of America, 3350 Scott Blvd., Bldg. 34, Santa Clara, CA 95054.
Note: This work was done when the first author was on leave from UCLA and doing a summer job at AT&T Laboratories.
†AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974.

provers effectively. Model checking is automatic, but is applicable to only the verification of finite-state systems, and suffer from the state-explosion problem, which inhibits its use in very big systems. Combining the two formal verification methods is appealing, as it has the potential of exploiting the strengths of both methods.

Prior research in this direction, e.g. [9, 11, 17], has focused on how to decompose a verification problem into parts each of which is manageable by one of the two methods. Although some impressive case studies have been conducted, it is still not clear how practical is the combination of theorem proving and model checking in general. The first problem is that some "real-world" systems may not allow any useful decomposition at all. And, secondly, even if a useful decomposition does exist, it may be very hard in practice to find it.

We suggest here another possible approach to combining mechanical theorem proving and model checking: we use mechanical theorem proving to formally verify the *meta-theory* of model checking. By "meta-theory" we mean the mathematical arguments justifying the correctness of model checking algorithms. These arguments can be quite complicated, and are used to justify complex algorithms. Erroneous proofs of algorithms and, more seriously, seemingly correct proofs of incorrect algorithms, are not uncommon in the literature. Using mechanical theorem proving to formally verify the meta-theory, we hope to raise the standard of rigor of, and hence our confidence in, model checking. Furthermore, since the meta-theory of a model checking algorithm needs to be verified only once while the algorithm can be used many times, the high cost of mechanical theorem proving can be amortized. We hope this will make mechanical theorem proving more attractive to practitioners of verification.

As a case study, we use the mechanical theorem prover HOL [5] to verify a *partial-order reduction* technique for cutting down the amount of state search performed by model checkers [16]. We choose this example for two reasons. First, this reduction technique has been implemented in the protocol analysis tool SPIN [7, 8] to significantly speed up the analysis of many practical protocols; hence its correctness has important practical consequences. Second, the correctness arguments involve nontrivial mathematics about sequences, relations, Mazurkiewicz traces [14], deterministic transition systems, and their interactions, whose formal versions we hope will become the basis of a formal meta-theory of many model checking algorithms and techniques.

At this point, a skeptic may ask: how do we know that HOL is free of bugs? And, if HOL turns out to contain bugs, what is the point of this whole exercise? Indeed, we do not know whether HOL is free of bugs. But, as we will explain later, HOL is built in such a way that its correctness depends only on a small kernel of carefully constructed code, which we believe is less error-prone than some model checking programs. To answer the second question, we note that we gained better understanding of the meta-

theory from the very act of formalization. Indeed, we were able to obtain a nontrivial generalization of the original informal theory (see Section 2.1 for details). The final formal proof is more elegant and better organized than the original informal proof, and certainly contains less (if any at all!) errors. In general, we believe that formalization is an invaluable tool for gaining better understanding of mathematical arguments.

The rest of this paper is organized as follows. Section 2 describes the partial-order reduction technique to be verified. Section 3 introduces higher-order logic (the logic supported by HOL) and mechanical theorem proving in HOL. Section 4 outlines our proof of the partial-order reduction technique in HOL. Section 5 discusses both encouraging and discouraging lessons learned from this exercise.

2 Partial-Order Reduction

Partial-order reduction is a technique that exploits the commutativity between independent actions of concurrent programs to reduce the space and time requirements of model checking. In the simplest case, consider a program in which two concurrent processes P_1 and P_2 can execute two actions a_1 and a_2 independently, then the two executions a_1a_2 and a_2a_1 can be considered "equivalent" (the precise definition of this equivalence relation will be given shortly). Hence it is sufficient for a model checker to check only one of them. For programs with a high degree of concurrency, the savings thus gained can be considerable.

We shall model programs as *deterministic transition systems* (DTS's). Let A be a set of *actions* and S be a set of *states*. A DTS P is simply a *partial* function from $A \times S$ to S. For any $a \in A$ and $s \in S$, if $P(a, s)$ is defined, then a is *firable* at s and $P(a, s)$ equals the state resulting from *firing* a at s. The set of actions firable at s in P is denoted by $\mathcal{F}(P, s)$. Let A^*, A^ω, and A^∞ denote respectively the sets of finite, infinite, and all sequences over A (i.e., $A^\infty = A^* \cup A^\omega$). We can extend P to a partial function P^* from $A^* \times S$ to S in the obvious way: $P^*(\varepsilon, s) = s$ and $P^*(a^\frown w, s) = P^*(w, P(a, s))$, where ε denotes the empty sequence and $^\frown$ denotes sequence concatenation. (Note that $P^*(a^\frown w, s)$ is defined exactly when (a) $P(a, s)$ is defined and equals some state s' *and* (b) $P^*(w, s')$ is also defined.) A finite sequence w is *firable* at a state s iff $P^*(w, s)$ is defined; an infinite sequence is *firable* at s iff all of its finite prefixes are firable at s. A (finite or infinite) sequence that is firable at a state of P is called an *execution* of P. For any $s \in S$, the set of states *reachable* from s in P, denoted $\mathcal{R}(P, s)$, is the set $\{P^*(w, s) \mid w \text{ is finite and firable at } s \text{ in } P\}$.

We say that a DTS Q is a *sub-DTS* of a DTS P iff for any $a \in A$ and $s \in S$:

$$a \in \mathcal{F}(Q, s) \;\Rightarrow\; a \in \mathcal{F}(P, s) \wedge (Q(a, s) = P(a, s)) \qquad (1)$$

Note that every execution of Q is an execution of P, but not necessarily vice versa.

An *independence relation* for P is an irreflexive and symmetric relation $\perp \subseteq A \times A$ such that for any $a \perp b$ and $s \in S$,

$$a \text{ is firable at } s \;\Rightarrow\; (b \text{ is firable at } s \Leftrightarrow b \text{ is firable at } P(a,s)) \tag{2}$$

$$a \text{ and } b \text{ are firable at } s \;\Rightarrow\; (P^*(a^\frown b, s) = P^*(b^\frown a, s)) \tag{3}$$

The equivalence relation between executions mentioned above is called *trace equivalence*, which was originally proposed for finite sequences by Mazurkiewicz [14] and later extended to infinite sequences by Kwiatkowska [12]. Trace equivalence is defined in five stages:

T1. The *1-step relation* $\doteq \subseteq A^* \times A^*$:
$v \doteq w$ iff there exist actions a and b and finite sequences x and y such that $a \perp b$, $v = x^\frown a^\frown b^\frown y$, and $w = x^\frown b^\frown a^\frown y$.

T2. The *finite trace equivalence relation* $\equiv^f \subseteq A^* \times A^*$:
$\equiv^f$ is the reflexive and transitive closure of $\doteq$.

T3. The *finite trace precedence relation* $\sqsubseteq^f \subseteq A^* \times A^*$:
$v \sqsubseteq^f w$ iff there exists a finite sequence u such that $w \equiv^f v^\frown u$.

T4. The *trace precedence relation* $\sqsubseteq \subseteq A^\infty \times A^\infty$:
$v \sqsubseteq w$ iff for each finite prefix x of v, there exists a finite prefix y of w such that $x \sqsubseteq^f y$.

T5. The *trace equivalence relation* $\equiv \subseteq A^\infty \times A^\infty$:
$v \equiv w$ iff $v \sqsubseteq w$ and $w \sqsubseteq v$.

An execution w of P is *fair* [12] iff whenever an action a is firable at some state s in w (i.e., firable after some finite prefix of w is fired), some action b dependent on a (i.e., $a \not\perp b$) must appear later than s in w.

Let Q be the reduced sub-DTS of P, to be constructed by the partial-order reduction algorithm. Let $s_0 \in S$ be an *initial state*. Three conditions will be imposed on Q:

C0. For any $s \in \mathcal{R}(Q, s_0)$, if $\mathcal{F}(P,s) \neq \emptyset$, then $\mathcal{F}(Q,s) \neq \emptyset$.

C1. For any $s \in \mathcal{R}(Q, s_0)$ and finite sequence w firable at s in P, if every action in w belongs to $A \setminus \mathcal{F}(Q,s)$, then every action in w is independent of every action in $\mathcal{F}(Q,s)$.

C2. There exists a *well-founded* relation $\gg \subseteq S \times S$ (i.e., there is *no* infinite descending chain $t_0 \gg t_1 \gg t_2 \gg \cdots$ in S) such that for any $s \in \mathcal{R}(Q, s_0)$, if $\mathcal{F}(Q,s)$ is a *proper* subset of $\mathcal{F}(P,s)$, then $s \gg Q(a,s)$ for every $a \in \mathcal{F}(Q,s)$.

We are now ready to state the main theorems:

Theorem 1 Suppose **C0** and **C1** are true. For any $s \in \mathcal{R}(Q, s_0)$ and $w \in A^\infty \setminus \{\varepsilon\}$ such that w is a *fair* execution of P from s, there exists $a \in \mathcal{F}(Q, s)$ such that $a \sqsubseteq w$.

This theorem essentially says that although Q may not be able to fire w at s, Q can fire some action a at s, resulting in a state s', from which the DTS P can still complete a sequence which is trace equivalent to w (notice that s' is a state of both P and Q). However, this theorem does not guarantee that by repeatedly taking such actions, the sub-DTS Q can generate an execution that is trace equivalent to w. To do so, we need the next theorem, which uses **C2**.

Theorem 2 Suppose **C0**, **C1**, and **C2** are true. For any $s \in \mathcal{R}(Q, s_0)$, $a \in A$, and $w \in A^\infty$ such that $a^\frown w$ is a *fair* execution of P from s, there exists $v \in A^*$ such that $v^\frown a$ is firable at s in Q, a is independent of every action in v, and $v^\frown a \equiv a^\frown v \sqsubseteq a^\frown w$.

Theorem 1 merely guarantees that Q won't be stuck at s. Theorem 2 is stronger: it guarantees that Q eventually can fire what P can fire at s. It is proved by well-founded induction on $\gg$, which exists because of **C2**. Finally, the following theorem justifies partial-order reduction:

Theorem 3 Suppose **C0**, **C1**, and **C2** are true. For any *fair* execution $w \in A^\infty$ of P from s_0, there exists an execution v of Q from s_0 such that $v \equiv w$.

This theorem says that if we partition the (fair) executions of P into equivalence classes of $\equiv$, then the executions of Q must contain at least one representative from each class. A property C is *trace equivalence robust* (or just *robust*) iff:

$$v \equiv w \;\Rightarrow\; (v \in C \Leftrightarrow w \in C) \tag{4}$$

Therefore, if the property to be checked is trace equivalence robust, then a model checker needs to check only the executions of Q in order to know whether C holds for all executions of P. If P has a high degree of concurrency, Q can be considerably less expensive to check than P.

Because trace equivalent executions differ only in the order of concurrent events, most "natural" properties are robust. For example, both fairness [12] and firability are robust properties. In case the property to be checked is not robust, one can try to refine the trace equivalence relation (by using a smaller independence relation on actions) until the property is robust with respect to it [16]. But, of course, the finer the trace equivalence relation is, the less savings one is likely to get from partial-order reduction.

2.1 *Comparison with the original paper*

In [16], a modified depth-first search (DFS) algorithm for constructing Q from P (which must be finite) is given. There, a much less abstract version of **C2** in terms of the DFS is used:

C2-DFS. For any s that is about to be closed by the DFS, $\mathcal{F}(Q,s)$ is allowed to be a *proper* subset of $\mathcal{F}(P,s)$ only when $Q(a,s)$ is not on the search stack for each $a \in \mathcal{F}(Q,s)$; otherwise, $\mathcal{F}(Q,s)$ must equal $\mathcal{F}(P,s)$.

To see that **C2-DFS** implies **C2**, let us number the states of Q according to the order in which they are closed by the DFS. Clearly, such a numbering induces a well-founded relation on states. Furthermore, it is a well-known property of the DFS that if a state t is reached from s during the DFS while t is not on the search stack, then t will be closed before s. Hence, t will be numbered with a smaller value than s.

Note that *any* assignment of ordinals to a set of objects induces a well-founded relation on these objects. Conversely, any well-founded relation can be obtained in this manner. Therefore, an equivalent formulation of **C2** is:

C2-Ord. There is an assignment $\mathcal{O}$ of ordinals to states in S such that for any $s \in \mathcal{R}(Q,s_0)$, if $\mathcal{F}(Q,s)$ is a *proper* subset of $\mathcal{F}(P,s)$, then $\mathcal{O}(s) > \mathcal{O}(Q(a,s))$ for every $a \in \mathcal{F}(Q,s)$.

We did not use **C2-Ord** for the technical reason that it is hard to deal with ordinals in HOL.[1]

The more abstract condition **C2** used here shows that DFS is not the only way to construct Q from P. For example, one can use breadth-first search (BFS) as follows: one starts by estimating an upper bound N of the number of states of P. Then the initial state gets the number N and each successive level of the BFS gets a number that is one less than the previous level. A state $s \in \mathcal{R}(Q,s_0)$ will either have $\mathcal{F}(Q,s) = \mathcal{F}(P,s)$, or have all actions in $\mathcal{F}(Q,s)$ leading to states in the next level of the BFS (all of which, incidentally, have the same number).

So our efforts in formalizing (parts of) [16] actually produced a genuine generalization of the theory in [16]: we now know that DFS is not the only algorithm compatible with partial-order reduction. Another (though, from the point of view of model checking, less useful) generalization is that **C2** and hence Theorem 3 are applicable to infinite-state systems as well.

The paper [16] contains actually four partial-order reduction algorithms: P can have or have no fairness assumption, and the construction of Q can be

[1] Intuitively, each type in higher-order logic is a set, while the collection of all ordinals is a proper class. So it is impossible to have a "type of ordinals" in higher-order logic (though it is possible to show in higher-order logic that every type can be well-ordered).

carried out before ("off-line") or during ("on-the-fly") the model checking of Q. We have formally verified only the simplest case: fair and off-line. The verification of the other three algorithms needs additional formal theories, such as those about stuttering equivalence [13], Levi Lemma for traces [14], and finite automata on infinite sequences [18]. The work required to develop those formal theories is significant, but not very different from what we have already done.

3 Higher-Order Logic and HOL

Higher-order logic, also known as the *simple theory of types*, was introduced by Alonzo Church [2] as a foundation of mathematics. It is sufficiently expressive to allow the formalization of virtually all of "ordinary mathematics" as *definitional extensions* of a handful of axioms and primitive inference rules. The HOL system, a mechanization of higher-order logic developed by Gordon and others [5], supports this definitional approach to formalizing mathematics by making a clear distinction between *definitions*, which can never introduce any inconsistency, and arbitrary *axioms*, which may introduce inconsistencies and hence should not be used casually. (For an eloquent objection to the casual use of axioms in verification, see the preface of [1].) As is the case in almost all works using HOL, we have not found it necessary to introduce one single axiom in this work: all notions needed are formalized as definitions and all properties desired are proved as theorems of those definitions.

Higher-order logic generalizes first-order logic by allowing quantification over functions. For example, the induction principle for natural numbers can be expressed in higher-order logic by a single formula:

$$\forall P\,.\,P(0) \wedge (\forall n\,.\,P(n) \Rightarrow P(n{+}1)) \Rightarrow (\forall n\,.\,P(n)) \qquad (5)$$

Every term t in higher-order logic has a type τ, written $t : \tau$. Intuitively, t denotes a value belonging to the set of values denoted by τ. For example, in (5), we have:

$$\begin{array}{ll} n : \mathit{num} & + : \mathit{num} \rightarrow \mathit{num} \rightarrow \mathit{num} \\ P : \mathit{num} \rightarrow \mathit{bool} & \wedge : \mathit{bool} \rightarrow \mathit{bool} \rightarrow \mathit{bool} \end{array}$$

where *num* is the type of natural numbers 0, 1, 2, ..., *bool* is the type of truth values T and F, and $\alpha \rightarrow \beta$ is the type of functions from α to β, for any types α and β. Note that a formula is just a term of type *bool*. A function $f : \alpha \rightarrow \beta$ can be applied to an argument $a : \alpha'$ only when α and α' are identical; the result $f(a)$ is of type β. For any variable $v : \alpha$ and any term $t : \beta$, the *lambda abstraction* $(\lambda v\,.\,t)$ denotes a function of type $\alpha \rightarrow \beta$ which, when applied to an argument $a : \alpha$, returns $t[a/v]$, the result of substituting a for v in t. (Note that v need not actually occur in t.)

Primitive terms of higher-order logic are of two kinds: *constants* and *variables*. A constant denotes a fixed value of the appropriate type, while a variable denotes an arbitrary value and hence can be bound by a quantifier. For example, in (5), 0, +, $\Rightarrow$ are constants, and n and P are (bound) variables. (There is no free variable in (5).) New constants can be introduced using *constant definitions*. For example, the definition:

$$\mathsf{sos}(x)(y) \quad \triangleq \quad x * x + y * y$$

defines a constant $\mathsf{sos} : num \rightarrow num \rightarrow num$ that computes the sum of squares of its arguments.

New types of *literals* can be defined using Melham's *recursive type definition* package [15] (though all literal types used in this paper are in fact non-recursive). For example, Melham's package can take a "BNF" of the form:

$$(\alpha, \beta)ttt \quad ::= \quad \mathtt{aaa} \mid \mathtt{bbb}(\alpha) \mid \mathtt{ccc}(\beta)$$

and define a new *parameterized* type $(\alpha, \beta)ttt$ with three constructors:

$$\mathtt{aaa} : (\alpha, \beta)ttt \qquad \mathtt{bbb} : \alpha \rightarrow (\alpha, \beta)ttt \qquad \mathtt{ccc} : \beta \rightarrow (\alpha, \beta)ttt$$

such that every value of type $(\alpha, \beta)ttt$ equals exactly one of $\mathtt{aaa}$, $\mathtt{bbb}(x)$, and $\mathtt{ccc}(y)$, for some unique x or y. The ability to have parameterized types such as $(\alpha, \beta)ttt$, which can be instantiated by substituting specific types for the *type variables* α and β, is called *type polymorphism* and is an enhancement made by HOL to the original formulation of Church.

There are other definitional mechanisms in HOL in addition to constant definition and recursive type definition, but they will not be needed in this paper.

HOL is implemented using a strongly typed functional programming language ML (for "Meta Language").[2] In HOL, the terms and theorems of higher-order logic are represented by two distinct ML types `term` and `thm`. The only constructors of `thm` are the mechanizations of the primitive inference rules of higher-order logic. (Note that axioms can be viewed as degenerate primitive inference rules.) For example, the primitive inference rule *Modus Ponens*:

$$\frac{\vdash a \Rightarrow b \quad \vdash a}{\vdash b}$$

is implemented in HOL as an ML function `MP:thm->thm->thm` that, if supplied with an implicational theorem `th1` and another theorem `th2` that is α-convertible to the antecedent of `th1`, returns the consequent of `th1` as a theorem (otherwise, `MP` raises an exception). Since ML is strongly typed,

[2] In fact, ML was invented for the very purpose of implementing LCF [6], a very influential mechanical theorem prover of which HOL is a descendant.

the only way one can generate an object of type `thm` is by repeated application of these primitive inference rules. (In particular, one cannot "cast" an arbitrary `term` into a `thm`.[3]) This is how HOL ensures the soundness of deductions it makes, and why the correctness of HOL depends only on the small kernel of ML codes implementing the primitive inference rules.

Although every `thm` generated by HOL is ultimately constructed using primitive inference rules, one seldom works directly at such a low level of abstraction when proving theorems in HOL. Since ML is a general-purpose programming language, sophisticated proof strategies, such as forward chaining, backward chaining, rewriting, and decision procedures for decidable theories, can be, and indeed many have been, programmed in HOL by combining primitive inference rules to form *derived proof procedures*. One can perform thousands of primitive inferences with a single invocation of a derived proof procedure. So HOL is an extremely malleable system whose capability is limited only by the user's imagination and programming ability and by the hardware resources of the machine on which HOL is run.

The typographic convention used to write higher-order logic types and terms in this section will be followed in the sequel: type variables are denoted by lowercase Greek letters, type constants by *slanted* identifiers, variables by *italic* identifiers, constants by sans serif identifiers or non-alphabetic symbols, and literals by `typewriter` identifiers. Free variables in a definition or theorem are implicitly universally quantified.

4 Verification of Partial-Order Reduction in HOL

Our verification of partial-order reduction in HOL is organized as a collection of *theories*, each of which consists of a set of definitions and a set of theorems together with their proofs. The dependency relation between the theories is depicted in Figure 1, where $\mathtt{S} \rightarrow \mathtt{T}$ means that theory `T` depends upon theory `S`. The contents of these theories are outlined below.

MISC: This theory proves miscellaneous theorems about natural numbers, lists, conditionals, and so on, that are needed in other theories. It contains no definition.

NUM: This is a theory about a new type *Num* of natural numbers augmented with an infinity element:

$$Num \quad ::= \quad \mathtt{Fin}(num) \mid \mathtt{Inf}$$

where *num* is the (built-in) type of natural numbers. Values of type *Num* are used to denote the "lengths" of sequences (see the theory

[3] Some implementations of HOL do contain a function `mk_thm` for doing just that. Clearly, no proof that uses `mk_thm` should be trusted. Our proof does not use `mk_thm`.

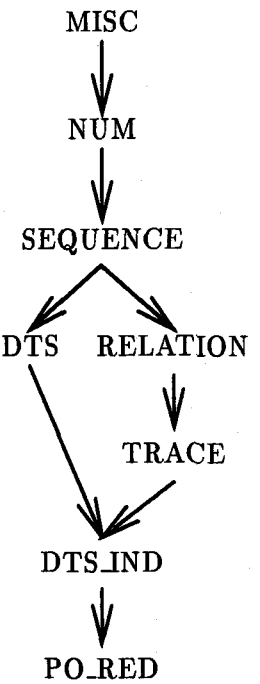

FIGURE 1. Dependencies between our theories

SEQUENCE). Also defined in this theory are arithmetic operations and ordering on *Num* and the least upper bound of any subset of *Num* (which always exists due to the presence of **Inf**).

SEQUENCE: This is a theory about a new type $(\alpha)Seq$ of *sequences* over an arbitrary type α, which is essentially the disjoint union of the type $(\alpha)list$ of lists over α, which represent finite sequences, and the type $num \rightarrow \alpha$ of functions with domain *num* and codomain α, which represent infinite sequences:

$$(\alpha)Seq \quad ::= \quad \mathtt{FinSeq}((\alpha)list) \mid \mathtt{InfSeq}(num \rightarrow \alpha)$$

The main reasons for having both finite and infinite sequences are that program executions can be both finite and infinite, and that some operations on sequences (such as projection mentioned below) can be more naturally defined. This theory is in fact the largest of our theories and contains definitions and theorems about numerous accessories, constructors, and relations on sequences, such as indexing, concatenation, prefix ordering, and the limit of an ascending chain of sequences (according to the prefix ordering). The *projection* of a sequence $s : (\alpha)Seq$ on a set $p : \alpha \rightarrow bool$ (we identify a set with its characteristic function), denoted $s \downarrow p$, is the sequence obtained by removing from s all elements that do not satisfy p. Note that projection can produce both finite and infinite sequences from an infinite sequence. Projection is defined using limits of projections of finite sequences, which are in turn defined using recursion on lists.

RELATION: This is a theory of *relations*, i.e., objects whose types are of the form $\alpha \rightarrow \alpha \rightarrow bool$. Various properties of relations (e.g., reflexivity, symmetry, transitivity) and operations on relations (e.g., reflexive and transitive closure) are defined. The following is a representative theorem of this theory, which is used several times in the theory **TRACE**:

$$\forall f : \alpha \rightarrow \beta \,.\, \forall R\, S \,.$$
$$\mathsf{Homomorphism}(f)(R)(S) \;\Rightarrow\; \mathsf{Homomorphism}(f)(R^*)(S^*)$$

where R^* and S^* are the reflexive and transitive closures of relations R and S, and Homomorphism is defined by:

$$\mathsf{Homomorphism}(f)(R)(S) \;\triangleq\; \forall x\, y \,.\, R(x)(y) \;\Rightarrow\; S(f(x))(f(y))$$

Also proved in this theory is the principle of well-founded induction.

TRACE: This is a theory of Mazurkiewicz traces. As outlined in Section 2, we define trace equivalence and precedence in five steps **T1**–**T5**. We

then prove a useful alternative characterization of trace precedence and equivalence:

$$s \sqsubseteq t \;\Leftrightarrow\; \forall a\, b.\, a \not\perp b \Rightarrow (s \downarrow \{a,b\} \preceq t \downarrow \{a,b\})$$
$$s \equiv t \;\Leftrightarrow\; \forall a\, b.\, a \not\perp b \Rightarrow (s \downarrow \{a,b\} = t \downarrow \{a,b\})$$

where $\perp$ is the underlying independence relation between actions and $\preceq$ is the prefix relation between sequences. Numerous properties about trace equivalence and precedence are proved.

DTS: This is a theory of *deterministic transition systems* (DTS's), which represent programs. A DTS is a function of type $\alpha \rightarrow \sigma \rightarrow (\sigma)Lift$, where α and σ are respectively the types of actions and states, and $(\sigma)Lift$ is σ augmented with a bottom element representing undefinedness:

$$(\sigma)Lift \;::=\; \mathtt{Up}(\sigma) \mid \mathtt{Bottom}$$

For any DTS P, action a, and state s, if $P(a)(s) = \mathtt{Bottom}$ then a is not *firable* at s in P; otherwise, $\mathsf{Down}(P(a)(s))$ is the state resulting from firing a at s in P, where $\mathsf{Down}(\mathtt{Up}(s)) \triangleq s$. The bulk of this theory is concerned with generalizing the notions of firability and firing from single actions to sequences.

DTS_IND: This is a theory of the interaction between deterministic transition systems and Mazurkiewicz traces. The main definitions are the compatibility of a DTS P with an independence relation (i.e., (2) and (3) of Section 2) and the notion of fairness. The main theorems proved are the robustness of firability and fairness with respect to trace equivalence.

PO_RED: This is where **C0–C2** and Theorems 1–3 of Section 2 are formalized and verified.

Table 1 gives some statistics about our theories: the numbers of definitions, main theorems, lines of code, and primitive inferences performed per theory and per line of code. The last number can be seen as a rough measure of the "density" of each theory. The whole verification project was carried out in 10 weeks of intense programming, but the final ML code takes less than five minutes to run on an SGI Challenger machine with 12 processors and more than 1 gigabytes of physical memory.

5 Lessons Learned

In this section, we shall attempt to answer the following questions: How hard is it to do a formal verification of a nontrivial model checking algorithm? Do the benefits justify the costs? Do we gain better understanding

Theory	Defs.	Thms.	Lines	Inferences	Infs./Line
MISC	0	43	524	15,368	29.33
NUM	14	21	477	9,345	19.59
SEQUENCE	30	128	2,520	107,176	42.53
RELATION	19	28	534	11,904	22.29
TRACE	10	103	1,639	36,679	22.39
DTS	10	21	400	8,563	21.41
DTS_IND	2	22	446	16,889	37.87
PO_RED	5	4	889	53,631	60.33
Total	90	370	7,429	259,555	34.94

TABLE 1. Statistics about our theories

of the verified algorithm? Do we learn something new about the verified algorithm? Is the formal proof more readable than the original informal one?

5.1 Encouraging Lessons

Generalizing the verified algorithm. Doing a formal proof requires mathematical abstraction. This process often produces new insights into the proved problem or algorithm. Defined in mathematical terms, the underlying assumptions and mathematical structures behind the algorithms become clear. In some cases, one realizes that it is possible to generalize or improve the original algorithm. This happened during our verification effort. In order to simplify the proof, the use of DFS in generating the state space was abstracted away and an ordinal numbering of the states was introduced in its stead. This numbering corresponds in DFS to the order in which the search backtracks from nodes. Originally, condition **C2** refers specifically to the DFS stack. In the formal proof it was abstracted into a requirement on an ordinal numbering. The more abstract requirement fits not only DFS but also other search mechanisms, e.g., BFS. This is useful, as sometimes one prefers BFS to DFS in model checking. For example, COSPAN [10] uses BFS because it simplifies the search for the shortest counterexample when a program does not satisfy its specification.

Gaining experience in techniques and tools. The verification attempt was also aimed at learning from the process of doing the verification on a "real-life" problem. This includes: experiencing with the formal verification tool HOL, learning its strengths and weaknesses, and studying how difficult it is to give a formalization of the verification problem in a given logic. Doing the proof, we have studied the feasibility of such a process. This includes the time required to complete such a proof, the level of expertise in the verification tool, the mathematical theories needed, the length of the

proof, and the kind of hardware that can carry such a verification task. The main lesson is that such a proof is feasible but requires non-trivial human resources in terms of expertise and time. One of the lessons we learned is the relation between the formal and informal proof. Going through the process of formal verification with a tool such as HOL forced us to be honest in our argumentation. Incantations such as "trivial", "obvious", "immediate from the definition", and "left as an exercise for the reader" do not work. The formal verification has helped to clean the proof and to identify points where "trivial" claims were in fact quite nontrivial.

Developing the machinery to verify similar problems. During the verification, mathematical theories that can support reasoning about related algorithms were developed. We have laid the groundwork of a formal infrastructure that will include the mathematical support for proving various automatic verification algorithms. Some of the theories described in Section 4 can be useful in verifying other model checking reduction algorithms, e.g., symmetry reduction [3, 4]. The theories outlined in Section 4 are not sufficient by themselves, as other algorithms may rely on other theories, e.g., permutation groups. We hope that the combined effort of several such proofs can create the infrastructure for verifying many model checking algorithms. Ideally, after collecting some additional theories related to model checking such as theories of ω-regular languages and automata, verifying a new algorithm would take much less time.

Experimenting with verifying a nontrivial algorithm. Software verification has lost some of its appeal over the last decade, as it was applied mainly to toy examples. In this experiment it was applied to a nontrivial algorithm, whose correctness relies on various mathematical theories. Moreover, the verification was made on a high level description of the algorithm, proving a generic method that can be implemented in many ways rather than just one instance of it. The high level description of the algorithm fits well with the HOL ability to reason about general mathematics and higher order objects. We hope to have demonstrated that formal verification is a practical and useful tool.

Gaining more confidence in an algorithm. Since a lot of effort is spent in creating software verification tools, it is all the more important to pay attention to their correctness. While a verification tool which reports nonexistent bugs ("false negatives") merely wastes time, a verification tool that does not report existing bugs is dangerous, as it creates unjustified confidence. A hand written proof attached to an algorithm is not always sufficient. In many scientific publications the proof of theorems is omitted or only sketched. The algorithm we study here is intricate enough not to be trusted on an intuitive basis. For example, there is a subtle point concerning why condition **C2** is needed when checking liveness properties.

5.2 *Discouraging Lessons*

The question of the correctness of the verifier. A software tool of the size of HOL is likely to contain some bugs. The ones that concern us are those that allow untrue theorems to be proved. In the past, such bugs have been found in HOL. Although the chance of actually encountering such a bug is insignificantly small, it demonstrates that the use of a formal tool does not guarantee the correctness of a proof. However, one can argue that the additional reliability is mainly achieved by the act of formalization. This argument should be taken with a grain of salt in light of the amount of automation in such tools. For example, in HOL and similar tools, the use of powerful rewriting procedures might produce a big deduction step, which might not always be easy to trace by the person who does the verification.

The possibility of making an error in the formalization of the problem. This seems to be the most severe problem. The first step of the verification is formalizing the theorem to be verified. The use of a specific logic such as higher-order logic requires the translation of the verification problem from a natural language into a formal language. This step is prone to errors. Errors in translating the supporting lemmas for the proof may be discovered when they cannot be proved or when they are insufficient to prove the main theorems.

A partial verification. Verifying algorithms can be done at various levels. One can verify the mathematical theorems that justifies an algorithm, an abstract algorithm given in pseudo-code, or a real implementation of the algorithm in a programming language with a formal semantics. The more abstract proof will be useful for more than one implementation, but would allow a higher level of uncertainty about the correctness of any particular implementation. This work was done at a rather abstract level, concentrating on the mathematical issues.

Poor readability of the proof. Although we gained a lot of new mathematical insights during the development of the HOL proof, the end product is no clearer than the proof in the original paper: the HOL proof code is very difficult to read. It is impractical to replace the informal proof in the paper by the formal HOL proof. Even an expert of both HOL *and* the verified problem can find it difficult to understand the intuition behind the proof given as HOL code. HOL does not impose any proof style which would make the proof a readable document.

The formal proof requires a lot of human resources. The verification effort resulted in almost 7,500 lines of dense proof code. Moreover, it required the complete attention of an HOL expert for 10 weeks. As pointed out in Section 2.1, only a portion of [16] was verified. Although we intended

initially to verify a bigger portion of [16], we found it impossible to achieve that goal in the given time frame. We hope that combining several such projects would create a more advanced starting point for future verification efforts.

Acknowledgements

We would like to thank Elsa Gunter for her penetrating remarks (especially regarding the formalization of deterministic transition systems), and Rajeev Alur, Gerard Holzmann, Doug Howe, Robert Kurshan, and Mihalis Yannakakis for their helpful comments.

6 References

[1] Robert S. Boyer and J Strother Moore, *A Computational Logic*, Academic Press, 1979.

[2] Alonzo Church, "A Formulation of the Simple Theory of Types", in *Journal of Symbolic Logic*, Vol. 5, pp. 56–68, 1940.

[3] E.M. Clarke, T. Filkorn, S. Jha, Exploiting Symmetry in Temporal Logic Model Checking, 5th International Conference on Computer Aided Verification, Elounda, Greece, June 1993, LNCS 697, 450–462.

[4] E.A. Emerson, A.P. Sistla, Symmetry and Model Checking, 5th International Conference on Computer Aided Verification, Elounda, Greece, June 1993, LNCS 697, 463–479.

[5] Michael J.C. Gordon and Thomas F. Melham (Ed.), *Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic*, Cambridge University Press, 1993.

[6] M.J.C. Gordon, A.J.R.G. Milner, and C.P. Wadsworth, *Edinburgh LCF: A Mechanized Logic of Computation*, Lecture Notes in Computer Science 78, Springer-Verlag, 1979.

[7] G.J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, 1991.

[8] G.J. Holzmann, D. Peled, An Improvement in Formal Verification, *7th* International Conference on Formal Description Techniques, Berne, Switzerland, 1994, 177–194.

[9] H. Hungar, "Combining Model Checking and Theorem Proving to Verify Parallel Processes", 5th International Conference on Computer Aided Verification, Elounda, Greece, June/July 1993, LNCS 697, pp.154–165.

[10] R.P. Kurshan, Computer-Aided Verification of Coordinating Processes, Princeton University Press, 1994.

[11] R.P. Kurshan and L. Lamport, "Verification of a Multiplier: 64 Bits and Beyond", 5th International Conference on Computer Aided Verification, Elounda, Greece, June/July 1993, LNCS 697, pp.166–179.

[12] M. Z. Kwiatkowska, Event Fairness and Non-Interleaving Concurrency, Formal Aspects of Computing 1 (1989), 213–228.

[13] L. Lamport, What good is temporal logic, in R.E.A. Mason (ed.), Proceedings of IFIP Congress, North Holland, 1983, 657–668.

[14] A. Mazurkiewicz, Trace Theory, in: W. Brauer, W. Reisig, G. Rozenberg (eds.) Advances in Petri Nets 1986, Bad Honnef, Germany, Lecture Notes in Computer Science 255, Springer, 1987, 279–324.

[15] Thomas F. Melham, "Automating Recursive Type Definitions in Higher-Order Logic", pp. 341–386 of G. Birtwistle and P.A. Subrahmanyam (Ed.), *Current Trends in Hardware Verification and Automated Theorem Proving*, Springer-Verlag, 1989.

[16] D. Peled, Combining Partial Order Reductions with On-the-fly Model-Checking, 6th International Conference on Computer Aided Verification, Stanford, CA, USA, 1994, LNCS 818, 377–390.

[17] S. Rajan, N. Shankar, M.K. Srivas, An Integration of Model Checking with Automated Proof Checking, 7th International Conference on Computer Aided Verification, Liège, Belgium, July 1995, LNCS 939, 84–97.

[18] Wolfgang Thomas, "Automata on Infinite Objects", pp.133–192 of Jan van Leeuwen (Ed.), *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics*, The MIT Press/Elsevier, 1990.