Taking the hol out of HOL

Nancy A. Day
Oregon Graduate Institute
Portland, OR, USA
nday@cse.ogi.edu

Michael R. Donat and Jeffrey J. Joyce
Intrepid Critical Software Inc.
Vancouver, BC, Canada
{Michael.Donat, Jeffrey.Joyce}@intrepid-cs.com

Technical Report CSE-00-003
Department of Computer Science
Oregon Graduate Institute

January 31, 2000

Abstract

We describe a systematic approach to building tools for the analysis of specifications expressed in higher-order logic (hol) outside the framework of a conventional, interactive theorem proving environment. Tools such as HOL and PVS integrate the tasks of parsing and typechecking a hol specification with substantial and complex theorem-proving functionality. In contrast, we have taken "the hol out of HOL" by building automated tools on top of just a parser and type-checker to eliminate the burden of the skilled interaction required in a conventional theorem prover. Our lightweight approach allows a hol specification to be used for diverse purposes such as refutation-based analysis and the algorithmic generation of test cases. Our toolset contains a variety of general-purpose utilities for the manipulation of higher-order logic specifications. For example, one utility allows a hol specification to be "evaluated" in a manner similar to the evaluation of a functional program. These utilities are combined to implement analysis techniques such as symbolic model checking. After five years of experience with this approach, we conclude that by decoupling hol from its conventional environment, we retain the benefits of an expressive specification notation, and can generate many useful analysis results automatically.

1 Introduction

Formal methods have come a long way. Industrial standards such as IEC 61508, and DO-178B include explicit references to the use of formal methods as a means of increasing confidence in safety-related systems. Formal methods add precision and checkability to various aspects of the system development process.

A decade ago, there was a wide chasm between specialized automated methods such as model checking [5], specification-intensive methods such as the use of Z [31], and general proof-based reasoning found in tools such as HOL [15]. The input notations of the analysis tools matched the analysis capabilities of the tool. For example, the SMV [25] notation describes finite state machines, whereas the use of higher-order logic (hol)[1] as the specification language of PVS corresponds to the intended use of PVS [27] as an interactive theorem-prover.

Progress is being made rapidly on bridging this chasm and uniting the capabilities of the various tools under one roof. For example, the SCR toolset includes a consistency checker, a simulator, a link to a model checker, and a link to a theorem prover [19, 2]. PVS has many automated decision procedures on call [26]. Most of these examples are, however, either application-specific such as the SCR toolset, or start from a heavyweight theorem prover.

We have been exploring a different point in the design space of these combined systems. For the past five years, in an industry/university collaborative research project, we have used hol as a specification notation and applied automated analysis techniques such as refutation-based ap-

[1] We will use "hol" or "Hol" for higher-order logic by itself, and "HOL" to refer to the HOL theorem proving system.

proaches (i.e., those that generate counterexamples), and test generation to these specifications. We have taken "the hol out of HOL" by building these automated procedures on top of just a parser and typechecker to eliminate the burden of the skilled interaction required in a conventional theorem prover.

The combination of hol with automated analysis may seem crippled from the beginning: we do not have all the tools we might need to work with our specification. However, our experience shows that less power is often better. The expressivity of higher-order logic allows us to embed more familiar notations within hol. The difficulties for new users come when the only tool support available has a high learning curve, and they struggle to understand the feedback the tool provides them about their specification. We offer a solution that lessens the learning curve, delaying the need to use a theorem prover until the problem requires it and the user is ready for it.

In Sections 2, and 3 we present our reasons for choosing to work with higher-order logic outside of a theorem proving environment. In Section 4, we describe our toolset, a collection of cooperating utilities that manipulate hol expressions in "truth-preserving" ways, i.e., the result of every transformation could also have been produced by a formal derivation using inference rules in HOL. In Section 5, we describe how the blocks are used in combination to construct analysis procedures such as symbolic model checking, and test generation.

Unlike our related presentations of this project [23, 9, 8, 7, 13], in this paper we focus on the capabilities of the tool and how it is engineered. This paper is intended to be an "under the hood" look at building an analysis tool, illustrating how our toolkit facilitates significant reuse of components for diverse applications such as test generation and model checking. We have also created new analysis methods such as constraint-based simulation. Our focus on automated analysis forces us to provide the user with control of performance factors such as BDD [3] variable order. We have also created methods that allow us to maintain the information necessary to produce readable, traceable results given in terms of the original specification.

By providing a lightweight interface between a general-purpose notation and automated analysis, we offer a middle ground between special-purpose analysis tools and general-purpose theorem provers. Our goal is to bring the power of a range of automated analysis techniques to specifiers without sacrificing suitability and expressiveness of notation.

2 Why higher-order logic ?

Initially we chose higher-order logic as a specification notation independently of consideration for tool support. Our notation S [23] is a syntactic variant of the object language of the HOL theorem proving system. S was also influenced by Z in that it includes constructs for the declaration and definition of types and constants. It was developed to support the practical application of formal methods in industrial scale projects. In this section, we explain our motivation for choosing to work with S.

First, S is *general-purpose notation*; it does not impose any particular style of specification. We have used it to capture a stimulus-response style of specification. In other cases, we have embedded other notations such as statecharts [16], and tables in S [8, 1]. By placing specialized notations within a general-purpose environment, we can take advantage of many general-purpose features such as parameterization, and re-usable auxiliary definitions and infrastructure. In the specification of an aeronautical telecommunications network (ATN) written in our embedded statecharts style, we witnessed these benefits, which reduced the specification effort, and resulted in a more concise and readable specification [1]. Also, we do not have to repeat the effort of building analysis tools for particular notations. Once a notation is embedded in hol, many of our analysis tools can be applied.

Second, S is a logic. We have found that *uninterpreted constants* in a logic play a key role in allowing us to match the level of abstraction found in requirements specifications. Joyce has called uninterpreted constants, "a modern-day Occam's razor"[2] and points out their value in filtering non-essential details and in improving the readability of the specification [22]. Uninterpreted constants can be used to represent elements that have meaning to domain experts but whose definition is irrelevant for analysis of a requirements specification. For example, many air traffic control specifications depend on the

[2]The idea that the simplest approach should be used; essentially not adding details when they are unnecessary. (William of Occam, 1300 - 1349)

"flight level" of an aircraft. The details of how the flight level is determined may be irrelevant for analysis of some aspects of the system. The calculation of the "flight level" can be captured by an uninterpreted constant. Analysis results produced for a specification hold for any interpretation of the uninterpreted constants. While a final specification should be complete, the use of uninterpreted constants along the way allows some results to be produced without having to specify all the details.

Furthermore, a logic contains *quantifiers*, which often allow the expression of formal requirements to more closely correspond to their expression in natural language. Quantified statements can be used to capture domain knowledge that describes the environment of the specification. The ability to use a quantifier eliminates the need to spell out all cases where the assumption is relevant.

Third, S is expressive; while we will never be able to prove automatically every property of our specifications, our notation is unlikely to limit adding more automated analysis capabilities as they are developed.

3 Why not use a theorem prover ?

In our approach, we have focused on automated analysis of our specifications. There have been a variety of successful efforts to combine theorem provers with automated decision procedures, such as PVS. Our experience with HOL-Voss [24] suggest that having the theorem prover control the link to the decision procedures is not the optimal approach for automated analysis.

First, the infrastructure of the theorem prover is unnecessary for automated analysis and makes the approach clumsy and intimidating to the novice specifier. These difficulties are a factor in industry's resistance to formal methods. For example, we particularly wanted to avoid the need to learn a meta-language to accomplish the specification task. Therefore we made S the input language to our tool, and have very simple commands to invoke our analysis procedures. A second example is that rewriting by means of tactic application was used for expansion of definitions in HOL-Voss. This step was different for each specification analyzed. We have shown that an automatic technique, called symbolic functional evaluation, is sufficient for this task and requires no user expertise.

Second, theorem provers are verification-based analysis tools. The output of a theorem prover is confirmation of a conjecture. Often, more useful results of analysis are either evidence that refutes an interpretation of the requirements, or truth-preserving rearrangements of the specification in order to distill atomic behaviour. Refutation-based techniques produce a variety of results other than just theorems. For example, when analyzing a table for inconsistency, refutation-based techniques can clearly isolate the source of the inconsistency. Consequently, it is easier to interpret the result of a successful refutation attempt than a failed verification attempt. In using formal methods for an independent validation and verification effort, Easterbrook and Callahan abandoned the use of PVS to carry out completeness and consistency checks because of the difficulty of determining the source of an inconsistency in a failed proof [14].

Third, the results need to be expressed in terms of the original specification. In contrast to our approach, translating the specification for input to a specialized decision procedure results in output in terms of the translated version.

Theorem provers definitely have a role to play in the analysis of complex systems. We advocate an approach that complements the use of theorem provers because we work with the same notation. Novice users and experts can work side-by-side. We have a tool that translates our S specifications to input for the HOL theorem prover [23].

4 The Toolkit

Our toolkit consists of techniques that manipulate S expressions in truth-preserving ways. In this section, we describe the collection of techniques that are combined to build analysis procedures such as symbolic model checking. Figure 1 captures the architecture of our tool. In addition to the specification and commands, the input of semantic definitions allow the specifier to work with notations such as statecharts embedded in hol.

Some of the techniques, such as Boolean abstraction, can also be found in tools such as PVS. Others, such as symbolic functional evaluation (SFE) for expanding S expressions, we developed because we wanted to move outside the

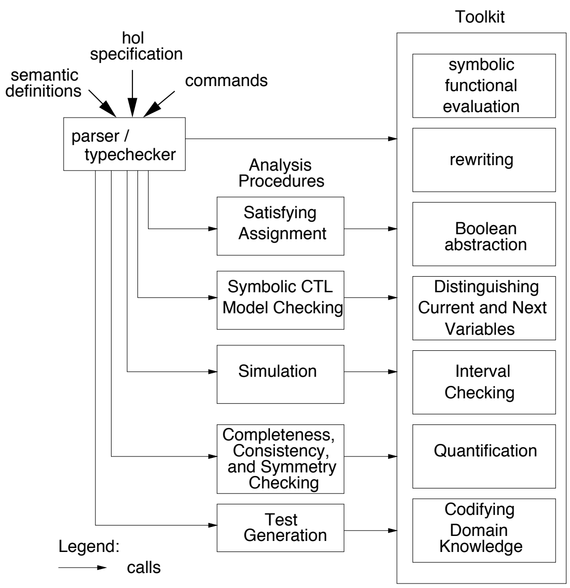

Figure 1: Architecture

theorem proving environment. In some cases, we rely on syntactic conventions for particular styles of specification. For example, we distinguish between the stimuli and response for test generation based on vocabulary conventions.

We also provide user access to performance tuning for some of these automated techniques. For example, while SFE is automatic, the user can control the depth of evaluation. For BDD-based analysis, we provide a way to input a variable order.

4.1 Symbolic Functional Evaluation

A specification consists of a collection of constant definitions, and declarations of types and constants. If we are using an embedded notation, then a set of semantic definitions are added to this collection. Often, the first step in analysis is to expand all of these definitions to determine the meaning of the specification.

Symbolic functional evaluation [7] (SFE) is a technique that we developed to "evaluate" or unfold S expressions, i.e., carry out the logical transformations of expanding definitions, beta-reduction, and simplification of built-in constants in the presence of quantifiers and uninterpreted constants. It extends mechanisms from functional language evaluation to carry out lazy evaluation of S expressions. Unlike using quote symbols in a language like LISP, SFE gives the user control over the depth of evaluation. For example, with the following declarations and definitions:

$$
\begin{aligned}
&z_1 : \text{num};\\
&f_1, f_2, f_3 : \text{num} \rightarrow \text{num};\\
&z_2 = f_1\ z_1;\\
&z_3 = f_2\ z_2;\\
&f_4(a) = f_3(a);
\end{aligned}
$$

The constants z_1, f_1, f_2, and f_3 are uninterpreted. When we evaluate the expression $f_4(z_3)$, we can instruct SFE to evaluate to one of three levels of evaluation. At the level of "complete" evaluation, it is expands all the definitions and returns the expression $f_3(f_2(f_1(z_1)))$. At the "point of distinction" level, SFE stops after it determines the tip of the expression is an uninterpreted function, and returns $f_3(z_3)$. One further level called "evaluated for rewriting" proved useful and evaluates the arguments of an uninterpreted function at the tip to the point of distinction. In this case, it would return $f_3(f_2(z_2))$. When abstracting an expression to propositional logic (see Section 4.3), the point of distinction level is most appropriate because any details revealed by evaluation are lost in abstraction.

Our implementation benefits from the use of a canonical syntactic representation of expressions, and caching of results.

SFE can be used to carry out symbolic simulation of specifications of hardware circuits as has been done previously in theorem provers e.g., [33, 32].

4.2 Rewriting

Once a specification has been sufficiently unfolded, several analyses require logical manipulation of the resulting hol formula. A rewrite toolkit component is useful for performing this task. For example, the following set of rewrite rules could be used to rewrite a specification so that negation ($\neg$) is only applied to predicates:

$$
\begin{gathered}
\forall X, Y. X \Rightarrow Y = \neg X \vee Y\\
\forall X, Y. \neg(X \wedge Y) = \neg X \vee \neg Y\\
\forall X, Y. \neg(X \vee Y) = \neg X \wedge \neg Y\\
\forall X. \neg\neg X = X\\
\forall P. \neg \forall x. Px = \exists x. \neg Px\\
\forall P. \neg \exists x. Px = \forall x. \neg Px
\end{gathered}
$$

Some analysis algorithms can be implemented as a series of rewriting operations. An example is the derivation of tests from an hol specification using a series of sets of rewrite rules [9, 12]. Implementing the test generator using rewriting is a better way to preserve logical soundness than an implementation as a series of ad-hoc manipulations.

Our lightweight rewrite system differs from some well-known rewrite systems, such as the one found in HOL. For performance reasons, the rewrite system cooperates with other means of simplification such as evaluating expressions with concrete values. The user of the rewrite system must ensure that each set of rewrite rules are confluent – otherwise, rewriting may not terminate. The user must also ensure that the rewrite rules are themselves sound. The checking of the rules need only be performed once as part of the development of an analysis procedure, and can be accomplished using a theorem-prover such as HOL or PVS.

Rewrite rules are stated as universally quantified equalities, e.g., $\forall x.E_1(x) = E_2(x)$, where x is a vector of variables. For rules specifying rewrites involving quantifiers and lambda abstraction:

1. variable capture is avoided using alpha conversion; and
2. if variable release occurs, the rewrite fails.

The concept of *variable release* is the opposite of variable capture. During rewriting, if a variable is quantified in an expression matching the left-hand side of the rewrite rule and is unquantified in the corresponding instance of the right-hand side, variable release has occurred. For example, applying $\forall P, Q.(\forall x.P \vee Q) = ((\forall x.P) \vee Q)$ to $\forall x.f\ x \vee y$ succeeds. However, applying the same rule to $\forall x.f\ x \vee g\ x$ fails because the x of $g\ x$ is released, i.e., x has become unquantified because it was free in Q. Rewrite rules requiring conditions on free variables can often be stated in terms of variable release.

The rewrite system also recognizes alpha equivalence, e.g., $(\lambda x.E(x)) = \lambda a.E(a)$.

The capabilities of variable release and alpha-equivalence allow many of the logical manipulation accomplished by conversions in a theorem prover to be performed by the rewrite system.

The rewrite system provides routines for applying a single rewrite to an expression, or to an expression and all its subexpressions. Sets of rules can also be applied. The depth of a rewrite operation can be limited by providing a call-back function that examines the current subexpression and signals the rewrite system to continue with this subexpression or go no deeper.

4.3 Abstraction to Propositional Logic

By abstracting our specifications to propositional logic, we can produce conservative analysis results automatically. As in Rajan [28], we decompose our S expression based on the logical operators of conjunction, disjunction, and negation. The fragments are assigned unique Boolean variables with alpha-equivalent subexpression matched to the same variable. We maintain a table matching the fragments to their Boolean variables to apply and reverse this process.

We also deal with enumerated types so that they are represented by multiple, related Boolean variables as in Ever [21]. Sections 4.5 and 4.7 discuss further ways of making this abstraction process less conservative.

Currently, we represent the expressions built from the Boolean variables using BDDs. A key to making this process efficient is to cache the match between hol expressions and BDD expressions. Once a BDD expression is created, an analysis procedure can manipulate it with the usual BDD package operations such as negation, conjunction, and quantification.

BDD variable order affects the size of the BDD representation of our hol expression. For small examples, it is sufficient to create the BDD variable as needed in the abstraction process, but for larger examples, a better method was required. In PVS, it is possible to request that dynamic variable order be carried out within the BDD package doing propositional simplification [30]. But, we found it critical to have direct support for providing the abstraction process with a BDD variable order to allow us to reuse a good order, as well as store and manipulate abstractions of expressions matched with constant names. Furthermore, we wanted the variable order stated in terms of expressions of the specification, not in terms of the Boolean variables that are substituted for those expressions during abstraction.

Therefore, we developed a way of supplying a variable ordering for BDDs as a list of S expressions. There are three types of substitutions: a single Boolean variable matched with a Boolean S expression, partitions discussed in Section 4.5, an enumerated types. Each type of substitution is accompanied by a list of numbers giving the position in the order of the Boolean variables used to represent the S expressions. We provide some utilities to help the user determine a good variable order by subcontracting the problem to existing verification tools such as the Voss Verification System [29]. Further details on our approach can be found in Day [6].

Creating a Boolean abstraction of an S expression and then reversing the process, can be a useful method of simplifying expressions that include quantification over Boolean variables. The resulting expression is logically equivalent to the original. Our tool provides a command that evaluates an expression to the desired level of evaluation using SFE, creates a BDD representation of the expression, and then creates an S expression from the BDD. We used this process in constructing a large next state relation by constructing conjuncts representing concurrent states individually first.

4.4 Distinguishing Current and Next Values

Specifications written in notations such as finite state machines describe a next state relation. Since hol has no built-in notion of dynamic behaviour, a means is required to distinguish the value of a variable in the current state from its value in the next. The following sections describe two approaches to this problem both based on syntactic conventions.

4.4.1 Variables as Functions

One approach is to make each variable a function mapping system states to values. This approach can be hidden from the user in an embedded notation. By explicitly representing the current and next states, and making variables be functions from states to values, we avoid the need to group the variables in a record structure explicitly as has been done in PVS [28].

For analysis procedures that iteratively explore the state space using the next state relation, such as symbolic model checking, and simulation, this element of the toolkit separates the Boolean variables representing the current state values and those for next state values. We use the syntactic convention that the variable cf represents the current state, and cf' the next state, thus a Boolean expression such as $x\ cf'$ refers to the value of the variable x in the next state. Care must be taken with expressions such as $x\ cf' = (x\ cf + 1)$ that are considered as one Boolean variable by the Boolean abstraction process, because they contain both cf and cf'. These expressions are treated as belonging to the next state.

4.4.2 Analysis-Guided Distinction

In some cases, the convention used to distinguish values in time is intrinsically linked the type of analysis. This is true of test generation, which guides the rewrite system to place expressions in certain forms to separate stimuli from responses. While this process is not technically not part of the toolkit, but rather is part of the specific analysis procedure, the two mechanisms that we describe here could be used in a similar manner to treating variables as functions mentioned in the previous section.

One mechanism is to base the distinction on whether variables within predicate arguments refer to the current or next state. As in Z, a prime (') is used to distinguish current state values from next state values. Thus, in the specification $(z = g(x, 5)) \Rightarrow \vee(z' = g(x, 10))$, $z = g(x, 5)$ refers to the current state because it does not contain a primed variable. The presence of z' indicates that $z' = g(x, 10)$ is a condition on the next state.

The second mechanism uses the syntactic convention that a literal beginning with a lower case letter indicates a next state predicate. A command can specifically label a literal as referring to either state, overriding this convention. This mechanism is appropriate in situations where the vocabulary used to specify next state values is different from that of specifying current state values, e.g., some applications of system-level requirements-based testing [13].

4.5 Interval Checking

The Boolean abstraction process is very conservative. Expressions such as $x < 5$, $(5 \leq x \wedge x \leq 10)$, and $10 < x$ should be abstracted to related Boolean expressions. One simple approach is to rewrite predicates involving inequalities into a canonical form to find relationships between expressions such as $x < 5$ and $5 > x$. However, this fails to capture the relationship between $x < 5$ and $x \leq 10$. Another approach is to abstract to a logic with a linear decision procedure. A third alternative is to use an external tool to add constraints based on the numeric relationships [4].

We chose a fourth approach that treats related expressions that partition a numeric value as an enumerated type. Based on known structure of a particular notation, we can identify some related expressions without a global search of the complete specification We encountered linear inequalities in tabular specifications where the cases in a table were based a partition of the range. A row of the table consisted of a list of expressions all relevant to the same numeric value. We can identify the row structure within the specification by searching for the `Row` keyword used in the embedding of the tabular notation.

To exploit the structure we extended our tool with a registry mechanism such that when certain keywords are encountered by SFE, particular procedures are carried out. The `Row` keyword is associated with a simple "interval checking" algorithm that takes the list of expressions in a row and determines if they represent a non-overlapping partition. In our current implementation, interval checking works for S expressions that contain relational numeric operators and have a concrete value on at least one side of the operator. Interval checking also returns any ranges not used in the row entries. This partition is used in the Boolean abstraction step of analysis to encode in Boolean values the related expressions that had previously been considered independent. Our registry mechanism makes it possible to extend easily SFE with other structure-specific rules.

4.6 Readable Results

A significant challenge in requirements analysis is returning results that are understandable and in the same terms as the specification despite the abstractions used in analysis. One step towards this goal is maintaining the information to reverse the Boolean abstraction as described in Section 4.3.

We are able to produce even better results by tracking information through the expansion and simplification processes of SFE and rewriting.

4.6.1 Unexpansion

Through an enhancement of the expression representation structure, we are able to return an expression in its unevaluated, and usually more compact form. Technically, lazy evaluation replaces a subexpression with its evaluated form, so the work of evaluation is done only once for all common subexpressions. We have modified our representation of expressions to include a pointer to the original, unevaluated version of the expression.

At the expense of memory, we are able to keep both the evaluated and unevaluated forms of the expressions during SFE. Some analysis procedures choose to output the unevaluated form of the expression to present a more abstract representation of the output.

4.6.2 Traceability

Unexpansion is not sufficient when manipulations other than expansion are performed. For analyses that perform rewriting, it is often critical to the user that the results be traceable to the source in the specification.

For example, tests generated from a specification are logical consequences of it. If a test is produced that represents clearly unintended behaviour, then its source in the specification needs to be located before it can be corrected.

An extension to our parser allows subexpressions within the hol specification to be tagged with user defined identifiers [10]. This use of identifiers is consistent with many requirements specification techniques now used in industry. During rewriting, the tags are propagated. By displaying these tags with the analysis results, the source of the results can be determined.

4.7 Quantification

Our specifications can include quantifiers. In abstraction, quantified subexpressions can be treated as single Boolean variable for the purpose of automated analysis. However, we can do a better than this conservative approach in certain circumstances. The substitutions and simplifications described in this section can either be done during SFE or rewriting, or as a separate function.

For quantified variables of types with a finite number of members we can substitute the possible values for the variable, e.g., universal quantification over a finite set of values can be expanded into a conjunction of conditions.

For quantified variables of infinite or uninterpreted types, we have methods for determining appropriate substitutions. For certain types of analysis such as completeness and consistency checking, and model checking, a conservative abstraction method is used to substitute any relevant uninterpreted constants into a universally quantified statement used in a negative positions, such as the antecedent of an expression. We found this form of substitution very useful for environmental assumptions, which are often stated with universal quantification. This reduction is conservative in that the abstracted expression is satisfied by more values than the original expression.

Another approach used in test generation is particularly interesting from a test coverage point of view. The user identifies the type of a quantified variable, treated as a set, as either static or dynamic. A type is *dynamic* if it can be different in different contexts of the specification. For example, quantification over the "flight" type might be dynamic, since there can be different numbers of aircraft within an airspace at any given time. An type is *static* if it is not dynamic, e.g., the set of natural numbers is a static specification element.

When a quantified variable has a type that is a dynamic set, we consider what instances of the type should be analyzed to ensure adequate coverage in testing. This type of simplification can be performed in at least three modes: none, single, or all. In the "single" mode of coverage, in the expression:

$$\forall x : X.\ P_1\, x \vee P_2\, x \vee \ldots \vee P_n\, x$$

we substitute a single value of type X, because this expression can be satisfied if one value has one of the properties P_i. In the "all" mode, we substitute n points, each one addressing a different P_i. Any constants introduced must be new, and free in the specification.

4.8 Codifying Domain Knowledge

Domain knowledge, or environmental assumptions, are conditions that must be taken into account during analysis to disregard infeasible combinations of conditions, and simplify expressions. In system-level requirements, we found there are relatively few dependencies between conditions, and therefore these can be expressed concisely using quantified axioms.

For some types of analysis, domain knowledge can be combined with the specification in analysis. It is the antecedent of the analysis goal, or conjuncted with the symbolic representation of the state set to enforce the constraint. In these cases, the substitution of relevant constants in the quantified expression described in Section 4.7 proved very useful.

In other types of analysis, such as test generation we cannot combine the statements of the domain knowledge with the specification because every part of the output must be traceable to the inputs. For these cases, we identified three schemata that capture the form of many of the axioms that are often used:

1. $\forall x.G \Rightarrow \mathrm{MutEx}[P_1\, x; P_2\, x; \ldots P_n\, x]$,

2. $\forall x.G \Rightarrow \mathrm{Subsm}[P_1\, x; P_2\, x; \ldots P_n\, x]$, and

3. $\forall x.G \Rightarrow \mathrm{States}[P_1\, x; P_2\, x; \ldots P_n\, x]$.

These schemata map the problem of simplifying an expression containing elements that match the patterns given in the schemata list to the problem of satisfying the guard G for the same instance of x. For example, conditions that form partial orders can be defined using Subsm. Conditions on the right subsume conditions on the left in the Subsm list. The statement $\forall x, y, z.x < y \Rightarrow \mathrm{Subsm}[x < z; y < z]$ captures the information that $y < z \Rightarrow x < z$ when $x < y$. The optional guard G, in this case $x < y$, provides a means of converting the dependency into a standard domain for which the analysis tool has a decision procedure. An expression such as $5 < x \wedge 10 < x$, is simplified by the schemata to $10 < x$ because it can check $5 < 10$. The MutEx form is used to define dependencies between mutually exclusive conditions. The States form defines conditions that represent a set of states; exactly one is true. These forms, combined with the pattern-matching capabilities provided by the rewrite system, are a powerful method of allowing the user to provide input to the tool as domain knowledge.

Though we found that the above approaches meet our needs, they have certain limitations. First, when there are more dependent relationships dictated by the environment, a formal model of the environment may be more concise than just axioms. Second, for more complex relationships it may be more efficient to provide a specially coded decision procedure, rather than pattern matching and basic evaluation to simply expressions.

5 Analysis Procedures

The procedures in our toolkit are combined together to form analysis procedures. In this section, we describe the procedures we have applied in examples. Table 1 is a partial list of the commands currently available in our tool.

5.1 Generating a Satisfying Assignment

To further one's understanding of the meaning of a complicated Boolean S expression, it can be useful to examine a satisfying assignment for that expression. This analysis procedure first expands the expression using symbolic functional evaluation, and then constructs a Boolean abstraction of the expression represented as a BDD. The user chooses the evaluation level for SFE. Using an algorithm found in the Voss system due to Carl Seger, we provide two possible ways of producing a satisfying assignment. One attempts to choose as many true assignments to variables as possible and the other has preference for false assignments.

5.2 Symbolic CTL Model Checking

Our model checking procedure takes constants with definitions that are 1) a CTL formula, 2) a next state relation, 3) an initial condition, and 4) an optional environmental constraint. We have a representation of CTL formula as an S datatype. Internally the expression representing the CTL formula is decomposed to invoke procedures based on the definitions of the component formulae. The next state relation, initial condition, and environmental constraint are all evaluated using SFE, and abstracted to propositional logic as a BDD. The current and next state variables are determined for the next state relation.

We currently have counterexample generation for AG and EF CTL formulae.

5.3 Simulation

For state machine specifications, a non-exhaustive form of configuration space exploration is simulation. The presence of uninterpreted constants in the specification forces our simulation to be symbolic.

Our analysis procedure does simulation based on the BDD representing the next state relation and constraint satisfaction. The user can constrain the set of assignments possible for the initial state and each subsequent state using a list of conditions. A particular assignment to the Boolean variables is chosen at each step. This assignment becomes the previous configuration for the next step. By choosing a particular assignment each time, this form of simulation does not encounter the state space explosion problem as in model checking.

A sequence of steps may not exist that satisfies the listed conditions. An arbitrary choice of a particular state that satisfies the constraints made early in the simulation may result in a satisfying sequence of steps not being found when one does exist. An alternative, slightly more expensive, analysis procedure carries out "one-lookahead". At each step, it chooses a configuration that satisfies the applicable constraint and has a next state that satisfies the next constraint in the list.

5.4 Completeness, Consistency, and Symmetry Checking

We use the same criteria as Heimdahl and Leveson [17], and Heitmeyer et al. [20] for the completeness and consistency of tabular specifications. Completeness analysis evaluates the expression consisting of the disjunction of the table's rows using SFE. After Boolean abstraction, we check if the expression is a tautology. If not, we reverse the abstraction, and use unexpansion to produce results in a column format indicating the cases that are not covered in the table. This presentation is possible because SFE maintains the unevaluated versions of expressions, and addresses some of the problems identified by Heimdahl in tracing the source of inconsistencies through nested tables where the output is completely expanded [18].

A similar procedure is carried out for consistency checking, where all pairs of columns are checked for overlap.

For symmetry checking, the analysis procedure constructs two versions of a two-parameter table with the parameters swapped, and checks if the two tables are the same.

Command	Action
`%setorder <const>`	use the BDD variable order given by the expression list `<const>`
`%save_bdd <const> <fname>`	save a BDD associated with a constant in the file
`%load_bdd <const> <fname>`	load a BDD from the file into constant
`%bddsimp <const> <ret_c>`	simplify `<const>` using BDDs; put result in `<ret_c>`
`%bddsatisfies <const>`	using BDDs, provide a satisfying assignment
`%ctlmc <ctl> <nsr> <ic> <env>`	do symbolic CTL model checking given next state relation, initial condition, and domain knowledge
`%simulate <nsr> <c_list>`	simulate the next state relation by satisfying the constraint list in each step
`%comp <const> <env>`	do completeness check of a tabular expression
`%cons <const> <env>`	do consistency check of a tabular expression
`%sym <const> <env>`	do symmetry check of a two-parameter tabular expression
`%tcg <options> <const>`	produce test classes and test frames for `<const>`

Table 1: Analysis Commands

5.5 Test Generation

System-level requirements-based test generation is an analysis that makes extensive use of rewriting. The rewrite rules used were verified using HOL. The hol specification is not assumed to be a state machine, but rather a relation between the stimuli and responses of the system.

After unfolding the specification to the desired level of detail, the resulting formula is transformed into its logically equivalent *Test Class Normal Form* (TCNF) [9, 12]. The TCNF is a conjunction of *test classes*, which describe particular stimulus/response behaviours as implications with the stimuli in the antecedent and responses in the consequent.

The antecedents of the test classes are rewritten further to reduce the size of quantified subexpressions. Choices (disjuncts) within an antecedent represent different test descriptions, referred to as *test frames*. A test frame is a test class that has no choice in the antecedent (other than instantiation). Domain knowledge is applied to simplify the test frames, and remove any that are infeasible.

Test frames are the results of the analysis, and are logical consequences of the given specification. Test frames are selected to cover the Boolean function represented by the test class antecedent using BDDs. The selection of test frames is determined by one of several coverage criteria chosen by the user.

6 Conclusions

We have described a lightweight approach for applying automated analysis techniques to higher-order logic specifications. To support this approach we have created utilities that that manipulate higher-order logic expressions in truth-preserving ways. These utilities handle the features of a logic, such as uninterpreted constants and quantification, in evaluation and abstraction.

We have demonstrated that a common core of utilities allows us to implement diverse analysis procedures such as test generation, and model checking. The common toolkit facilitates re-use of code and extension of the suite of analysis procedures with new methods such as symmetry checking and constraint-based simulation. We have also shown methods particular to embedded notations can be created such as the completeness and consistency analysis of tables.

Two other innovations of our approach are: we allow users to control performance factors such as BDDs in terms of the language of the specification; and through the analysis process we maintain information that produces readable, traceable results in the language of the specification.

Space does not permit us to describe the real-world examples that we have specified and analyzed using our tools. Examples include an aeronautical telecommunications network (ATN) [1, 6], a separation minima for aircraft [8, 11], a small heating system [6], a steam boiler control system [12], and parts of a proprietary air traffic management system [13]. These examples are non-trivial. For example, the parameterized formal ATN statechart specification is approximately 43 pages. The expanded S representation of the ATN next state relation consists of 52 076 nodes in a canonical form.

In the future, we would like to explore how other automated abstraction techniques can be used in our framework. For example, less conservative results can be achieved by abstracting to a variant of first-order logic. We would like to explore decomposition strategies to lessen the state space explosion problem. Our approach, which uses the same specification language as a high-powered tool where these strategies can be verified, allows experts to hard code their verification method to make it accessible to non-experts.

7 Acknowledgments

The first author is supported by Intel, NSF (EIA-98005542), USAF Air Materiel Command (F19628-96-C-0161), and the Natural Science and Engineering Research Council of Canada (NSERC). This paper is based on results produced by the formalWARE research project supported by the BC Advanced Systems Institute, Raytheon Systems Canada, and MacDonald Dettwiler. Details on the formalWARE research project can be found at `http://www.cs.ubc.ca/formalWARE`.

References

[1] J. H. Andrews, N. A. Day, and J. J. Joyce. Using a formal description technique to model aspects of a global air traffic telecommunications network. In *FORTE/PSTV*, 1997.

[2] M. M. Archer, C. L. Heitmeyer, and S. Sims. Tame: A PVS interface to simplify proofs for automata models. In *UITP'98*, 1998.

[3] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[4] W. Chan, R. Anderson, P. Beame, and D. Notkin. Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints. In *CAV*, volume 1254 of *LNCS*, pages 316–327, 1997.

[5] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[6] N. A. Day. *A Framework for Multi-Notation, Model-Oriented Requirements Analysis.* PhD thesis, Dept. of Comp. Sci, Univ. of British Columbia, 1998.

[7] N. A. Day and J. J. Joyce. Symbolic functional evaluation. In *TPHOLs*, number 1690 in LNCS, pages 341–358. Springer, 1999.

[8] N. A. Day, J. J. Joyce, and G. Pelletier. Formalization and analysis of the separation minima for aircraft in the North Atlantic Region. In *Lfm*, pages 35–49. NASA Conference Publication 3356, September 1997.

[9] M. R. Donat. Automating formal specification-based testing. In *TAPSOFT*, volume 1214 of *LNCS*. Springer, April 1997.

[10] M. R. Donat. Automatically generated test frames from a Q specification of ICAO flight plan form instructions. Technical Report 98-05, Dept. of Comp. Sci, Univ. of British Columbia, April 1998.

[11] M. R. Donat. Automatically generated test frames from an S specification of separation minima for the North Atlantic Region. Technical Report 98-04, Dept. of Comp. Sci, Univ. of British Columbia, April 1998.

[12] M. R. Donat. *A Discipline of Specification-Based Test Derivation.* PhD thesis, Department of Computer Science, University of British Columbia, 1998.

[13] M. R. Donat and J. J. Joyce. Applying an automated test description tool to testing based on system level requirements. In *INCOSE*, 1998.

[14] S. Easterbrook and J. Callahan. Formal methods for V & V of partial specifications: An experience report. In *RE*, pages 160–168, Annapolis, MD, 1997.

[15] M. Gordon and T. Melham, editors. *Introduction to HOL.* Cambridge University Press, 1993.

[16] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computing*, 8:231–274, 1987.

[17] M. P. Heimdahl and N. G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Trans. on Soft. Eng.*, 22(6):363–377, June 1996.

[18] M. P. E. Heimdahl. Experiences and lessons from the analysis of TCAS II. In *ISSTA*, pages 79–83, January 1996.

[19] C. Heitmeyer, J. Kirby, B. Labaw, and R. Bharadwaj. SCR*: A toolset for specifying and analyzing software requirements. In *CAV*, volume 1427 of *LNCS*, pages 526–531. Springer, 1998.

[20] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

[21] A. J. Hu, D. L. Dill, A. J. Drexler, and C. H. Yang. Higher-level specification and verification with BDDs. In *CAV*, volume 697 of *LNCS*. Springer, 1993.

[22] J. Joyce. *Multi-Level Verification of Microprocessor Based Systems*. PhD thesis, Cambridge Comp. Lab, 1989. Technical Report 195.

[23] J. Joyce, N. Day, and M. Donat. S: A machine readable specification notation based on higher order logic. In *International Workshop on the HOL Theorem Proving System and its Applications*, pages 285–299. Springer, 1994.

[24] J. Joyce and C.-J. Seger. Linking BDD-based symbolic evaluation to interactive theorem-proving. In *DAC*. IEEE Computer Press, 1993.

[25] K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, May 1992.

[26] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In *CAV*, 1996.

[27] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *CADE*, volume 607 of *LNCS*, pages 748–752, 1992.

[28] P. S. Rajan. *Transformations on Data Flow Graphs: Axiomatic Specification and Efficient Mechanical Verification*. PhD thesis, Dept. of Comp. Sci, Univ. of British Columbia, 1995.

[29] C.-J. H. Seger. Voss - a formal hardware verification system: User's guide. Technical Report 93-45, Dept. of Comp. Sci, Univ. of British Columbia, December 1993.

[30] N. Shankar, S. Owre, J. M. Rushby, and D. J. Stringer-Calvert. PVS prover guide, Sept. 1999. Version 2.3.

[31] J. Spivey. *Understanding Z*. Cambridge University Press, Cambridge, 1988.

[32] J. P. Van Tassel. A formalization of the VHDL simulation cycle. In *Higher Order Logic Theorem Proving and its Applications*, pages 359–374. North-Holland, 1993.

[33] P. J. Windley. *The Formal Verification of Generic Interpreters*. PhD thesis, University of California , Davis, 1990.