Symbolic Functional Evaluation

Nancy A. Day[1] and Jeffrey J. Joyce[2]

[1] Oregon Graduate Institute, Portland, OR, USA
nday@cse.ogi.edu
[2] Intrepid Critical Software Inc., Vancouver, BC, Canada
joyce@intrepid-cs.com

Abstract. Symbolic functional evaluation (SFE) is the extension of an algorithm for executing functional programs to evaluate expressions in higher-order logic. SFE carries out the logical transformations of expanding definitions, beta-reduction, and simplification of built-in constants in the presence of quantifiers and uninterpreted constants. We illustrate the use of symbolic functional evaluation as a "universal translator" for linking notations embedded in higher-order logic directly with automated analysis without using a theorem prover. SFE includes general criteria for when to stop evaluation of arguments to uninterpreted functions based on the type of analysis to be performed. SFE allows both a novice user and a theorem-proving expert to work on exactly the same specification. SFE could also be implemented in a theorem prover such as HOL as a powerful evaluation tactic for large expressions.

1 Introduction

Symbolic functional evaluation (SFE) is the extension of an algorithm for executing functional programs to evaluate expressions in higher-order logic. We use SFE to bridge the gap between high-level, expressive requirements notations and automated analysis by directly evaluating the semantics of a notation in higher-order logic. SFE produces the meaning of a specification in a form that can be subjected to behavioural analysis. In our approach, writing the semantics of a notation is the only step necessary to have access to a range of automated analysis procedures for specifications written in that notation. Bridging this gap allows a novice user to perform some types of analysis even if all queries of the specification cannot be checked automatically.

SFE carries out the logical transformations of expanding definitions, beta-reduction, and simplification of built-in constants in the presence of uninterpreted constants and quantifiers. While these logical transformations can be performed by rewriting in a theorem prover, they do not require the full generality of rewriting. To use an algorithm for evaluating functional programs, two special considerations are needed. First, we handle uninterpreted constants. Special treatment of uninterpreted constants allows us to carry out substitution, a core part of the evaluation algorithm, more efficiently. Quantifiers, such as "forall", are treated for the most part as uninterpreted constants. Second, we have defined several distinct and intuitive levels of evaluation that serve as "stopping points" for the SFE algorithm when considering how much to evaluate the arguments of uninterpreted functions. The choice of level provides the user with a degree of control over how much evaluation should be done and can be keyed to the type of automated analysis to be performed.

The following simple example illustrates the idea of symbolic functional evaluation. We use a syntactic variant of the HOL [17] formulation of higher-order logic called S [27].[1] In S, the assignment symbol `:=` is used to indicate that the function on the left-hand side is being defined in terms of the expression on the right-hand side. The function `exp` calculates x^y:

```
exp x y := if (y = 0) then 1 else (x * exp x (y - 1));
```

If we use SFE to evaluate this function with the value of `x` being 2 and the value of `y` being 3, the result is 8. In order to test the behaviour of `exp` for more possible inputs, we make the input `x` symbolic. The constant `a` is an uninterpreted constant. Evaluating `exp a 3`, SFE produces:

```
a * (a * (a * 1))
```

If both of the inputs to `exp` are uninterpreted constants, say `a`, and `b`, then the evaluation will never terminate. SFE can tell the evaluation may not terminate as soon as `exp` is expanded the first time because the argument to the conditional is symbolic, i.e., it cannot determine whether `b` is equal to zero or not. At this point SFE can terminate evaluation with the result:

```
if (b = 0) then 1 else (a * exp a (b - 1))
```

The conditional `if-then-else` is defined using pattern matching. This stopping point is the point at which it cannot determine which case of the definition of `if` to use. The conditional applied to this argument is treated as an uninterpreted function, i.e., as if we do not know its definition. We could continue to evaluate the arguments (`(b=0)`, `1`, and `(a * exp a (b - 1))`) further. In this case, continuing to evaluate the arguments indefinitely would result in non-termination. The levels of evaluation of SFE describe when to stop evaluating the arguments of uninterpreted functions. The levels of evaluation are general criteria that are applied across all parts of an expression.

Currently, within a theorem prover, evaluating an expression may require varying amounts of interaction by the user to choose the appropriate sequence of rewriting steps to obtain the desired expansion of the expression. With SFE, we make this expansion systematic using levels of evaluation. Thus, while being less general than rewriting, SFE provides precise control for the user to guide its specialised task. SFE also does not require the unification step needed for rewriting.

There are a variety of applications for symbolic functional evaluation. For example, Boyer and Moore [7] used a more restricted form of symbolic evaluation

[1] A brief explanation of the syntax of S can be found in the appendix.

as a step towards proving theorems in a first-order theory of lists for program verification.

In this paper, we show how SFE allows us to bridge the gap between high-level, expressive requirements notations and automated analysis. The input notations of many specialised analysis tools lack expressibility such as the ability to use uninterpreted types and constants, and parameterisation. One of the benefits of our approach is that it allows specifiers to use these features of higher-order logic in specifications written in notations such as statecharts [21], but still have the benefits of some automated analysis. Therefore, both a novice user and a theorem-proving expert can work on exactly the same specification using different tools. Section 8 provides an example of how general proof results can help the novice user in their analysis of a specification.

We bridge the gap between high-level notations and automated analysis by using SFE as the front-end to more specialised analysis tools. SFE directly evaluates the semantics of a notation in higher-order logic. It produces a representation of the meaning of the specification that can be subjected to behavioural analysis using automated techniques. Our framework is illustrated in Figure 1. SFE plays the role of a "universal translator" from textual representations of notations into expressions of the meaning of the specification.

Because the evaluation process need not always go as far as producing a completely evaluated expression to be sufficient for automated analysis, modes of SFE produce expressions at different levels of evaluation. Different abstractions are then applied for carrying out different kinds of automated analysis. For example, to use BDD-based analysis [8], we can stop evaluation once an uninterpreted function is found at the tip of an expression and then abstract to propositional logic. To use if-lifting, a particular type of rewriting, it is necessary to evaluate partly the arguments of an uninterpreted function.

By directly evaluating the formal semantic functions, our approach is more rigorous than a translator that has not been verified to match the semantics of the notation. Our framework is also more flexible than working in a current theorem proving environment. For example, we are able to return results in terms of the original specification and provide access to the user to control parameters such as BDD variable ordering.

Another application for symbolic functional evaluation is the symbolic simulation step used in many microprocessor verification techniques. Cohn [11], Joyce [26], and Windley [40] all used unfolding of definitions to execute opcodes in theorem proving-based verification efforts. More recently, Greve used symbolic simulation to test microcode [20]. The pipeline flushing approach of Burch and Dill begins with a symbolic simulation step [9]. Symbolic functional evaluation could be used inside or outside of a theorem prover to carry out this step for typed higher-order logic specifications. Inside a theorem prover, it could be used as a "super-duper tactic" [1].

The use of higher-order logic to create a formal specification can easily involve building up a hierarchy of several hundred declarations and definitions, including semantic functions for notations. SFE has been an effective tool in the analysis of

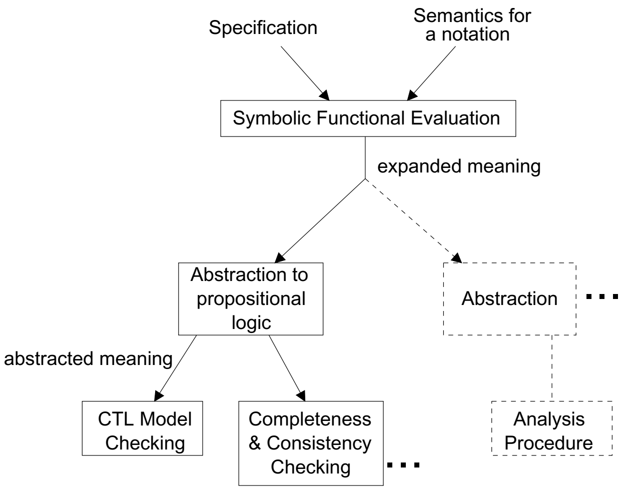

Fig. 1. Our framework

some large specifications. We have used our approach to analyse an aeronautical telecommunications network (ATN) written in a statecharts-variant embedded in higher-order logic. The specification consisted of approximately 3100 lines of text. The SFE step for the ATN took 111 seconds on an Ultra-Sparc 60 (300 MHz) with 1 GB RAM running SunOS 5.6 .

We present our algorithm in terms of expressions in the lambda calculus (applications, abstractions, variables). This presentation provides a simple interface for integrating our implementation of SFE with other tools, as well as giving enough details to implement SFE. SFE makes it possible to have a general-purpose specification notation as a front-end for many specialised analysis tools.

2 Related Work

Translation into the input notation of an existing analysis tool is a common approach to bridge the gap between specialised notations and analysis tools (e.g., [4,5,41]). There are three disadvantages to the translation approach. First, unless the translator has been verified, there is no assurance that the translator correctly implements the semantics of the notation. For notations such as statecharts, establishing the correctness of the translation is very difficult. Second, results are presented to the specifier in the terms of the translated specification, not the original specification. These results can be difficult to decipher. Third, translation often must include an abstraction step because the destination notation is unable to represent non-finite or uninterpreted constants. The translator then matches only a particular type of analysis.

Owre, Rushby, and Shankar have already demonstrated that the approach of embedding a notation in a general-purpose base formalism, such as the PVS form of higher-order logic, makes a range of analysis techniques accessible to a specification [31]. We show that this approach does not require theorem proving support. Besides being difficult to learn for a novice user, theorem provers are verification-based tools and often use decision procedures only for “yes/no” answers. They usually lack the ability to access counterexamples in terms of the original specification and do not allow the user to control the analysis with information such as BDD variable orderings. Our approach is more flexible because it operates independently of the analysis technique, and thus is unencumbered by the effort of dealing with layers of proof management often necessary to integrate decision procedures into theorem provers. This difficulty is noted in the work on integrating the Stanford Validity Checker (SVC) [25] with PVS [32].

It is possible to embed a notation in a functional programming language such as Haskell [33]. This approach has been used for creating domain-specific notations such as the hardware description language Hawk [29]. Symbolic data types can be used to represent uninterpreted constants but this approach requires explicitly describing a symbolic term structure [16]. Also, a programming language lacks a means of representing uninterpreted types and quantification.

There have been efforts to translate higher-order logic specifications into programming languages for execution [2,10,35]. These approaches have been limited to subsets of higher-order logic, often not including uninterpreted constants.

Boyer and Moore [7] used evaluation as a step in proving theorems in a first-order theory of lists for program verification. Their EVAL function is similar to SFE in that it deals with skolem constants. However, SFE also handles uninterpreted function symbols, which raises the question of how much to evaluate the arguments to these symbols. We provide a uniform treatment of this issue using levels of evaluation. These levels work for both uninterpreted functions, and functions such as the conditional defined via pattern matching. If the argument cannot be matched to a pattern, these functions are also treated as uninterpreted. SFE carries out lazy evaluation, whereas EVAL is strict.

Symbolic functional evaluation resembles on-line partial evaluation. The goal of partial evaluation is to produce a more efficient specialised program by carrying out some expansion of definitions based on static inputs [12]. With SFE, our goal is to expand a symbolic expression of interest for verification (i.e., all function calls are in-lined). For this application, all inputs are symbolic and therefore divergence of recursive functions in evaluation often occurs. Our levels of evaluation handle the problem of divergence. We have special treatment of uninterpreted constants in substitution because these are free variables. SFE also handles quantifiers and constants of uninterpreted types.

3 Embedding Notations

Gordon pioneered the technique of embedding notations in higher-order logic in order to study the notations [19]. Subsequent examples include a subset of the programming language SML [39], the process calculus value-passing CCS [30], and the VHDL hardware description language [38]. In previous work [13], we presented a semantics for statecharts in higher-order logic. We use an improved version of these semantics for analysing specifications written in statecharts in our framework. For model-oriented notations, such as statecharts, the embedding of the notation suitable for many types of automated analysis is an operational semantics that creates a next state relation. Symbolic functional evaluation works with both shallow and deep embeddings of notations [6].

4 A Simple Example

For illustration, we choose an example that is written in a simple decision table notation used previously in the context of carrying out completeness and consistency analysis [15]. Table 1 is a decision table describing the vertical separation required between two aircraft in the North Atlantic region. This table is derived from a document that has been used to implement air traffic control software. These tables are similar to AND/OR tables [28] in that each column represents a conjunction of expressions. If the conjunction is true, then the function `VerticalSeparationRequired` returns the value in the last row of the column. Expressions in columns are formed by substituting the row label into the underscore (later represented as a lambda abstraction). In this specification, the functions `FlightLevel`, and `IsSupersonic` are uninterpreted and act on elements of the uninterpreted type `flight`. They are declared as:

```
: flight;               /* declaration of an uninterpreted type */

FlightLevel : flight -> num;    /* declaration of uninterpreted */
IsSupersonic : flight -> bool;         /* constants */
```

We are interested in analysing this table for all instantiations of these functions. For example, we want to know if the table is consistent, i.e., does it indicate different amounts of separation for the same conditions? Specialised tools for carrying out completeness and consistency checking are based on notations that lack the ability to express uninterpreted constants and types (e.g., [22,23]). We use SFE as a universal translator to determine the meaning of a specification in this decision table notation. Using abstraction mechanisms, we can then convert the meaning of the table into a finite state form, suitable for input to automated analysis tools.

Table 1. Vertical separation

					Default
FlightLevel A	_ < = 280	.	_ > 450	_ > 450	
FlightLevel B	.	_ < = 280	_ > 450	_ > 450	
IsSupersonic A	.	.	_ = T	.	
IsSupersonic B	.	.	.	_ = T	
VerticalSeparationRequired(A,B)	1000	1000	4000	4000	2000

We represent tabular specifications in higher-order logic in a manner that allows us to capture the row-by-column structure of the tabular specification. The translation from a graphical representation into a textual representation does not involve interpreting the semantic content of the specification. Table 1 is represented in higher-order logic as:

```
VerticalSeparationRequired (A,B) := Table
[Row (FlightLevel A)  [(\x.x<=280); Dc ; (\x.x>450); (\x.x>450)];
 Row (FlightLevel B)  [Dc; (\x.x<=280); (\x.x>450); (\x.x>450)];
 Row (IsSupersonic A) [Dc; Dc; True; Dc];
 Row (IsSupersonic B) [Dc; Dc; Dc; True] ]
[1000; 1000; 4000; 4000; 2000];
```

`Dc` is “don’t care” replacing the “.” in the table. The notation `\x.` is a lambda abstraction. The syntax `[ ... ; ... ]` describes a list. In previous work, we gave a shallow embedding of this decision table notation in higher-order logic by providing definitions for the keywords of the notation such as `Row` and `Table`. We provide these definitions in the appendix.

The result of using SFE to evaluate the semantic definitions for the expression `VerticalSeparationRequired(A, B)` is:

```
if (FlightLevel A <= 280) then 1000
else if (FlightLevel B <= 280) then 1000
else if (FlightLevel A > 450) and (FlightLevel B > 450)
      and (IsSupersonic A) then 4000
else if (FlightLevel A > 450) and (FlightLevel B > 450)
      and (IsSupersonic B) then 4000
else 2000
```

The result of SFE is an expanded version of the meaning of the specification that can be subjected to behavioural analysis techniques. This result is semantically equivalent to the original specification. It is the same as would be produced by unfolding definitions and beta-reduction in a theorem prover, but we accomplish this without theorem proving infrastructure.

An abstracted version of the specification is usually needed for finite state automated analysis techniques. For completeness and consistency analysis, one suitable abstraction is to use a Boolean variable to represent each subexpression that does not contain logical connectives. For example, `FlightLevel B <= 280` is represented as a single Boolean variable. This process of abstracting to propositional logic is based on previous work by Rajan [36], and others. This abstraction is conservative in that the abstract version has more behaviours than the original specification.

5 Symbolic Functional Evaluation (SFE)

Our SFE algorithm is an extension of the spine unwinding algorithm for evaluation of functional programs found in Peyton Jones [34]. Functional programs are essentially the lambda calculus without free variables. Uninterpreted constants do not have definitions and are free variables in the lambda calculus.[2] We extend Peyton Jones' algorithm to include a case for variables with special treatment for the arguments of uninterpreted functions. For efficiency, we make a distinction between uninterpreted constants and other lambda calculus variables. We also introduce evaluation levels. Because the evaluation process need not always go as far as producing a completely evaluated expression to be sufficient for automated analysis, modes of SFE produce expressions at different levels of evaluation.

5.1 Levels of Evaluation

Levels of evaluation are based on the extent to which the arguments to uninterpreted constants, variables and data constructors in an expression are evaluated. The tip of an application is the leaf of the leftmost branch of an application, e.g., `f` for the expression `f a b c` as illustrated in Figure 2. Defined constants and abstractions at the tip are eliminated in evaluation.

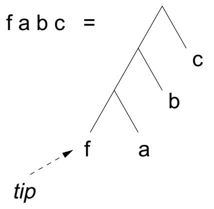

Fig. 2. Tip of a function application

We introduce three levels of evaluation: evaluated to the point of distinction (PD_EVAL), evaluated for rewriting (RW_EVAL), and completely evaluated (SYM_EVAL). These levels are ordered from the "least evaluated" to the "most evaluated". The desired level of evaluation is an input to the symbolic functional evaluation process. The user decides on the level of evaluation based on the type of automated analysis to be carried out. Here we present the levels informally, however a full specification of the levels of evaluation can be found in Day [14].

Evaluated to the Point of Distinction (PD_EVAL) If abstraction to propositional logic is used, evaluated to the point of distinction is sufficient, because the extra information exposed by further evaluation is lost in the abstraction process. For example, an expression such as:

```
IsOnRoute Routes1 A
```

with the declarations and definitions:

```
A : flight;

IsOnRoute : (location # location)set -> flight -> bool;

Routes1 :=
  {(USA,BDA);(CAN,BDA);(IberianPeninsula, Azores);
   (Iceland,Scandinavia);(Iceland, UnitedKingdom)};
```

[2] The G-machine [24], another implementation of graph reduction for evaluating functional programs, is not applicable here because of the free variables. It is necessary to avoid variable capture when doing substitutions of arguments that may contain free variables.

when completely evaluated becomes:

```
IsOnRoute {(USA,BDA);(CAN,BDA);(IberianPeninsula, Azores);
     (Iceland,Scandinavia);(Iceland, UnitedKingdom)} A
```

If subjected to abstraction to propositional logic, this whole expression is treated as one Boolean variable – expanding the definition of `Routes1` adds information that is lost in the abstraction process when replaced by a single variable. Therefore evaluation to the point of distinction only evaluates an expression to the point where the tip of the expression is an uninterpreted constant or data constructor.

Evaluated for Rewriting (RW_EVAL) To carry out "if-lifting", evaluation to the point of rewriting is needed. If-lifting is a method of rewriting expressions involving the conditional operator `if-then-else` to further reduce the expression. For example, if the value returned by the conditional expression is Boolean, then the following equality holds and can be used to eliminate the `if` function:

$$\texttt{if a then b else c} \equiv \texttt{(a and b) or (not(a) and c)} \qquad (1)$$

Jones et al. [25] describe "if-lifting" of expressions as a heuristic for their validity checking algorithm. They present two rules:[3]

$$\texttt{((if a then b else c) = (if a then d else e))} \equiv \texttt{if a then (b = d) else (c = e)}$$

$$\texttt{((if a then b else c) = d)} \equiv \texttt{if a then (b = d) else (c = d)}$$

We generalise these rules slightly to lift an argument with a conditional outside any uninterpreted function (not just equality). Eventually the expression may reach the point where equation (1) can be applied. Transforming an expression by if-lifting and then carrying out abstraction to propositional logic makes the abstraction less conservative. If-lifting can optionally be carried out during symbolic functional evaluation.

Evaluation to the point of rewriting evaluates each argument of an uninterpreted function to the point of distinction. This exposes conditional operators at the tip of the arguments so that if-lifting can be carried out. The `if` operator is used in the semantics of the decision table notation described in the simple example. Therefore, evaluation to the point of rewriting is often chosen for analysing decision tables.

Completely Evaluated (SYM_EVAL) Complete evaluation means all possible evaluation is carried out. The expression may still contain uninterpreted constants as well as some built-in constants of higher-order logic such as conjunction. Complete evaluation can be very helpful in checking the correctness of the semantics. It may also produce more succinct output than either of the other two levels, however, complete evaluation may not terminate.

[3] We use "if-then-else" rather than "ite".

5.2 Algorithm

SFE evaluates expressions in higher-order logic to the point where the expression is at a particular level of evaluation. The user chooses the mode for SFE usually based on the least amount of evaluation that is needed for the type of analysis to be carried out.

Although higher-order logic notation may be enhanced with various constructs, fundamentally it consists of just four kinds of expressions: 1) applications, 2) abstractions, 3) variables, and 4) constants. We subdivide the category of constants into: 4a) uninterpreted constants (including quantifiers), 4b) defined constants, 4c) data constructors, and 4d) built-in constants. Evaluation involves definition expansion, beta-reduction, and evaluation of built-in operations.

Our algorithm carries out normal order reduction, which means arguments to functions are not evaluated until they are used. Evaluation is carried out in place. Figure 3 gives the top-level algorithm in C-like pseudo code. It is called initially with an expression, an empty argument list, and the desired level of evaluation of the expression. In spine unwinding, the arguments to an application are placed on an expression list until the tip of the application is reached.

Compared to Peyton Jones' algorithm, we include extra checks at the beginning to stop evaluation if we have reached the correct level of evaluation. Some subexpressions may already have had some evaluation carried out on them because common subexpressions are represented only once. In this we differ from many interpreters and compilers for functional languages. Evaluation results are also cached.

We also include cases for variables and uninterpreted constants (a sub case of constants). In the cases for variables, uninterpreted constants, and early stopping points in evaluation, we have to recombine an expression with its arguments. Depending on the desired level of evaluation, `Recombine` may carry out more evaluation on the arguments of an uninterpreted function. For example, for evaluated for rewriting, the arguments are evaluated to the point of distinction.

The possible values for the "mode" parameter are elements of the ordered list SYM_EVAL, RW_EVAL, and PD_EVAL. All expressions begin with the level NOT_EVAL. An expression's evaluation tag is stored with the expression and is accessed using the function `EvalLevel`.

If the expression is an abstraction, the arguments are substituted for the parameters in the body of the lambda abstraction avoiding variable capture, and the resulting expression is evaluated. While uninterpreted constants are variables in the lambda calculus, we optimise substitution by treating them differently from other variables. By adding a flag to every expression to indicate if it has any variables that are not uninterpreted constants, we avoid walking over subexpressions that include uninterpreted constants but no parameter variables.

If the expression is an application, spine unwinding is carried out. After evaluation, the original expression is replaced with the evaluated expression, using the function `ReplaceExpr`. A pointer to the expression is used.

The evaluation of constant expressions (`EvalConstant`) is decomposed into cases. If the constant is a constructor or an uninterpreted constant, it is recom-

```
expr EvalExpression(expr *exp, expr_list arglist, level mode)

if (arglist==NULL) and (EvalLevel(exp) >= mode) then
    return exp
else if (EvalLevel(exp) >= mode) and (mode == PD_EVAL)  then
    return Recombine(exp,arglist,NOT_EVAL)

switch (formof(exp))
case VARIABLE (v) :
    if (arglist!=NULL) then return Recombine(exp, arglist, mode)
    else return exp
case ABSTRACTION (parem exp):
    (leftover_args,newexp) = Substitute(exp, parem, arglist)
    return EvalExpression(newexp,leftover_args,mode)
case APPLICATION (f a) :
    newarglist = add a to beginning of arglist
    newexp = EvalExpression(f, newarglist,mode)
    if (arglist==NULL) then ReplaceExpr(exp, newexp)
    return newexp
case CONSTANT(c) :
    return EvalConstant(exp, arglist, mode)
```

Fig. 3. Top-level algorithm for symbolic functional evaluation

bined with its arguments, which are evaluated to the desired level of evaluation. If the constant is a built-in function, the particular algorithm for the built-in constant is executed. If the expression is a constant defined by a non-pattern matching definition, it is treated as an abstraction. For a constant defined by a pattern matching definition, its first argument must be evaluated to the point of distinction and then compared with the constructors determining the possible branches of the definition. If a match is not found, the expression is recombined with its arguments as if it is an uninterpreted constant.

Some built-in constants have special significance. Conjunction, disjunction and negation are never stopping points for evaluation when they are at the tip of an expression because they are understood in all analysis procedures that we have implemented (completeness, consistency, and symmetry checking, model checking, and simulation). Our implementation of SFE can be easily extended with options to recognise particular uninterpreted constants (such as addition) and apply different rules for the level of evaluation of the constant's arguments and do some simplification of expressions. For example, SFE simplifies `1 + a + 1` to `2 + a`, where `a` is an uninterpreted constant.

Experiments using SFE as the first step to automated analysis revealed the value of presenting the user with the unevaluated form of expressions to interpret the results of the analysis. Evaluation in place implies the original expression is no longer available. However, by attaching an extra pointer field allowing the node to serve as a placeholder, the subexpressions of the old expression remain present. An option of SFE keeps the unevaluated versions of expressions.

No special provisions have been taken to check for nontermination of the evaluation process.

6 Quantifiers

Writing specifications in higher-order logic makes it possible to use quantifiers, which rarely appear in input notations for automated analysis tools. In this section, we describe two ways to deal with quantifiers that make information bound within a quantifier more accessible so that less information is lost in abstraction. These logical transformations can be optionally carried out during SFE.

First, we handle quantifiers over enumerated types. A simple enumerated type is one where the constructors do not take any arguments. If the variable of quantification is of a simple enumerated type, then the quantifier is eliminated by applying the inner expression to all possible values of the type. For example, using the definitions,

```
: chocolate := Cadburys | Hersheys | Rogers ; /* type definition */
tastesGood : chocolate -> bool;
```

the expression

```
forall (x:chocolate). tastesGood (x)
```

is evaluated to:

```
tastesGood (Cadburys) and tastesGood (Hersheys) and tastesGood (Rogers)
```

Second, specialisation (or universal instantiation) is a derived inference rule in HOL. Given a term t' and a term *forall* $x.t$ used in a negative position in the term (such as in the antecedent of an implication), the quantified term can be replaced by $t[t'/x]$, where t' replaces free occurrences of x in t. Our implementation has a “specialisation” option that carries out universal instantiation for any uninterpreted constants of the type of the quantified variable when universal quantification is encountered in a negative position in evaluation.

7 Abstraction

Symbolic functional evaluation produces an expression describing the meaning of a specification. After the SFE step, some form of abstraction is usually necessary for automated analysis. The abstraction depends on the type of analysis to be performed. We implemented an abstraction to propositional logic technique and represent the abstracted specification as a BDD. Together with SFE, this abstraction mechanism allows us to carry out completeness, consistency, and symmetry checking, model checking, and simulation analysis of specifications written in high-level notations.

When abstracting to propositional logic, subexpressions that do not contain Boolean operations at their tip are converted to fresh Boolean variables. For example, the statement,

```
(FlightLevel A > 450) and (FlightLevel B > 450) and (IsSupersonic A)
```

can be abstracted to,

```
x and y and z
```

with the Boolean variables `x`, `y`, and `z` being substituted for the terms in the original expression, e.g., `x` for `(FlightLevel A > 450)`. This is a conservative abstraction, meaning the abstracted version will have more behaviours than the original specification. In this process, quantifiers are treated as any other uninterpreted constant. To present the output of analysis to the user, the abstraction process is reversed, allowing counterexamples to appear in terms of the input specification.

Along with reversing the abstraction process, the ability to keep the unevaluated versions of expressions during SFE means expressions can be output in their most abstract form. Keeping the unevaluated version of expressions also makes it possible to recognise structures in the specification that can help in choosing an abstraction. For example, the decision table form highlights range partitions for numeric values such as the flight level. In Day, Joyce, and Pelletier [15], we used this structure to create a finite partition of numeric values to produce more accurate analysis output.

Because BDD variable order is critical to the size of the BDD representation of the abstracted specification, we provide a way for the user to control directly this order. The user runs a procedure to determine expressions associated with Boolean variables in abstraction. They can then rearrange this list and provide a new variable order as input. We use a separate tool (Voss [37]) to determine suitable variable orders for our examples.

Other abstraction techniques could certainly be used. For example, we are considering what abstraction would be necessary to link the result of SFE with the decision procedure of the Stanford Validity Checker [25]. SVC handles quantifier-free, first-order logic with uninterpreted constants.

8 Analysing the Semantics of Notations

One of the benefits of our approach is that it allows specifiers to write in higher-order logic but still have the benefits of some automated analysis. Therefore, both a novice user and a theorem-proving expert can work on exactly the same specification using different tools. The theorem-proving expert might prove that certain manipulations that the novice user can do manually are valid with respect to the semantics of the notation. In this section, we give an example of such a manipulation.

We have used our approach to analyse an aeronautical telecommunications network (ATN) written in a statecharts-variant and higher-order logic [3]. The

ATN consists of approximately 680 transitions and 38 basic statechart states, and after abstraction is represented by 395 Boolean variables. The statechart semantics produce a next state relation. In these semantics, Boolean flags representing whether each transition is taken or not are existentially quantified at the top level. Turning the next state relation into a BDD required building the inner BDD and then doing this quantification. We discovered that the inner BDD was too big to build. Therefore we sought to move in the quantification as much as possible to reduce the intermediate sizes of the BDDs. The system was divided into seven concurrent components. We wanted to view the next state relation as the conjunction of the next state relations for each of the concurrent components by pushing the quantification of the transition flags into the component level. Gordon's work on combining theorem proving with BDDs addresses the same problem of reducing the scope of quantifiers to make smaller intermediate BDDs by using rewriting in HOL [18].

We hypothesise the following property of statecharts that could be checked in a theorem prover such as HOL: if the root state is an AND-state and each component state has the properties:

- no transition within a component is triggered in whole or in part by the event of entering or exiting a state that is not within the component
- the sets of names modified by actions for each component are disjoint

then:

$$\texttt{Sc (AndState [}st_1\texttt{;}st_2\texttt{; }\ldots st_n\texttt{])}\ \mathit{step} \equiv$$
$$\texttt{Sc}\ st_1\ \mathit{step}\quad \texttt{and}\quad \texttt{Sc}\ st_2\ \mathit{step}\quad \texttt{and}\quad \ldots\quad \texttt{and}\quad \texttt{Sc}\ st_n\ \mathit{step}$$

where Sc is the semantic function producing a next configuration relation, st_x is a component state, and *step* is a pair of configurations. (We use the term "configuration" to describe a mapping of names to values that is often called a "state".) We have sketched a proof of this property but have not yet mechanised this proof. This property was used with success to allow our tool to build the next state relation for the system and then carry out model checking analysis.

9 Conclusion

This paper has presented symbolic functional evaluation, an algorithm that carries out definition expansion and beta-reduction for expressions in higher-order logic that include uninterpreted constants. We have described the application of SFE to the problem of linking requirements notations with automated analysis. SFE acts as a universal translator that takes semantics of notations and a specification as input and produces the meaning of the specification. To connect a new notation with automated analysis, it is only necessary to write its semantics in higher-order logic. To use a new analysis procedure with existing notations, it is only necessary to provide the appropriate abstraction step from higher-order logic. By linking higher-order logic directly with automated analysis procedures, some queries can be checked of the specification, even if all types of queries cannot be processed automatically.

Our approach is more rigorous than a custom translator, and more flexible than working in a current theorem-proving environment. For example, we are able to return results in terms of the original specification and provide access to the user to control parameters such as BDD variable ordering.

Our algorithm for symbolic functional evaluation extends a spine unwinding algorithm to handle free variables with special considerations for uninterpreted constants. Levels of evaluation provide a systematic description of stopping points in evaluation to tailor the process to particular forms of automated analysis. Compared to an implementation of rewriting, SFE need not search a database for appropriate rewrite rules in a unification step, nor follow branches that are not used in the end result. SFE could be implemented within a theorem prover as a tactic.

Symbolic functional evaluation is applicable to any expression in higher-order logic. We plan to explore how SFE can be used stand-alone for symbolic simulation.

10 Acknowledgments

For helpful discussions and comments on this paper, we thank Byron Cook, Mark Jones, Dick Kieburtz, Jeff Lewis, and John Matthews of OGI, and the reviewers. 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 the first author's PhD research, which was supported by the BC Advanced Systems Institute, Raytheon Systems Canada, and MacDonald Dettwiller.

References

1. M. D. Aagaard, M. E. Leeser, and P. J. Windley. Towards a super duper hardware tactic. In J. J. Joyce and C.-J. H. Seger, editors, *HOL User's Group Workshop*. Springer-Verlag, August 1993.
2. J. H. Andrews. Executing formal specifications by translating to higher order logic programming. In *TPHOLs*, volume 1275 of *LNCS*, pages 17–32. Springer Verlag, 1997.
3. 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'97*, 1997.
4. J. M. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *IEEE Trans. on Soft. Eng.*, 19(1):24–40, January 1993.
5. R. Bharadwaj and C. Heitmeyer. Verifying SCR requirements specifications using state space exploration. In *Proceedings of the First ACM SIGPLAN Workshop on Automatic Analysis of Software*, 1997.
6. R. Boulton et al. Experience with embedding hardware description languages in HOL. In *Theorem Provers in Circuit Design*, pages 129–156. North-Holland, 1992.

7. R. S. Boyer and J. S. Moore. Proving theorems about LISP functions. *Jour. of the ACM*, 72(1):129–144, January 1975.
8. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
9. J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *CAV*, volume 818 of *LNCS*, pages 68–79. Springer-Verlag, 1994.
10. A. J. Camilleri. Simulation as an aid to verification using the HOL theorem prover. Technical Report 150, University of Cambridge Computer Laboratory, October 1988.
11. A. Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5(2):127–139, June 1989.
12. C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *POPL*, pages 493–501. ACM Press, 1993.
13. N. Day and J. Joyce. The semantics of statecharts in HOL. In *HOL User's Group Workshop*, volume 78, pages 338–351. Springer-Verlag, 1993.
14. N. A. Day. *A Framework for Multi-Notation, Model-Oriented Requirements Analysis.* PhD thesis, Department of Computer Science, University of British Columbia, 1998.
15. N. A. Day, J. J. Joyce, and G. Pelletier. Formalization and analysis of the separation minima for aircraft in the North Atlantic Region. In *Fourth NASA Langley Formal Methods Workshop*, pages 35–49. NASA Conference Publication 3356, September 1997.
16. N. A. Day, J. R. Lewis, and B. Cook. Symbolic simulation of microprocessor models using type classes in Haskell. To appear in CHARME'99 poster session.
17. M. Gordon and T. Melham, editors. *Introduction to HOL.* Cambridge University Press, 1993.
18. M. J. C. Gordon. Combining deductive theorem proving with symbolic state enumeration. Transparencies from a presentation, summer 1998.
19. M. J. C. Gordon. Mechanizing programming logics in higher-order logic. In *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 387–439. Springer-Verlag, 1988.
20. D. A. Greve. Symbolic simulation of the JEM1 microprocessor. In *FMCAD*, volume 1522 of *LNCS*, pages 321–333. Springer, 1998.
21. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computing*, 8:231–274, 1987.
22. 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.
23. 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.
24. T. Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of the ACM Conference on Compiler Construction*, pages 122–32, 1984.
25. R. B. Jones, D. L. Dill, and J. R. Burch. Efficient validity checking for processor verification. In *ICCAD*, 1995.
26. J. Joyce. *Multi-Level Verification of Microprocessor Based Systems.* PhD thesis, Cambridge Comp. Lab, 1989. Technical Report 195.
27. 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, 1994.
28. N. G. Leveson et al. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.

29. J. Matthews, B. Cook, and J. Launchbury. Microprocessor specification in Hawk. In *International Conference on Computer Languages*, 1998.
30. M. Nesi. Value-passing CCS in HOL. In *Higher Order Logic Theorem Proving and Its Applications*, pages 352–365. LNCS 780, Springer-Verlag, 1993.
31. S. Owre, J. Rushby, and N. Shankar. Analyzing tabular and state-transition requirements specifications in PVS. Technical Report CSL-95-12, Computer Science Laboratory, SRI International, April 1996.
32. D. Y. Park, J. U. Skakkebæk, M. P. Heimdahl, B. J. Czerny, , and D. L. Dill. Checking properties of safety critical specifications using efficient decision procedures. In *FMSP'98*, 1998.
33. J. Peterson and K. Hammond, editors. *Report on the Programming Language Haskell.* Yale University, Department of Computer Science, RR-1106, 1997.
34. S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* International Series in Computer Science. Prentice Hall, 1987.
35. P. S. Rajan. Executing HOL specifications: Towards an evaluation semantics for classical higher order logic. In *Higher Order Logic Theorem Proving and its Applications*, pages 527–535. North-Holland, 1993.
36. P. S. Rajan. *Transformations on Data Flow Graphs.* PhD thesis, University of British Columbia, 1995.
37. C.-J. H. Seger. Voss - a formal hardware verification system: User's guide. Technical Report 93-45, Department of Computer Science, University of British Columbia, December 1993.
38. 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.
39. M. VanInwegan and E. Gunter. HOL-ML. In *Higher Order Logic Theorem Proving and Its Applications*, pages 61–74. LNCS 780, Springer-Verlag, 1993.
40. P. J. Windley. *The Formal Verification of Generic Interpreters.* PhD thesis, University of California, Davis, 1990.
41. P. Zave and M. Jackson. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology*, 2(4):379–411, October 1993.

A Semantics for the Decision Table Notation

This appendix appeared previously in Day, Joyce, and Pelletier [15].

The S notation is very similar to the syntax for the term language used in the HOL theorem prover. But unlike HOL, S does not involve a meta-language as part of the specification format for declarations and definitions. Instead, the syntax for declarations and definitions is an extension of the syntax used for logical expressions. (In this respect, S more closely resembles Z and other similar formal specification notations.) For example, the symbol `s := f` is used in S for a definition, e.g., `TWO := 2`, in contrast to an assertion, e.g., `TWO = 2`.

Another difference that will likely be noticed by readers familiar with HOL is the explicit type parameterisation of constant declarations and definitions. Type parameters, if any, are given in a parenthesised list which prefixes the rest of the declaration or definition. This is illustrated in the definitions given below by the parameterisation of `EveryAux` by a single type parameter, `ty`.

Many of the definitions shown below are given recursively based on the recursive definition (not shown here) of the polymorphic type `list`. These recursive definitions are given in a pattern matching style (similar to how recursive functions may be defined in Standard ML) with one clause for the `NIL` constructor (i.e., the non-recursive case) and another clause for the `CONS` constructor (i.e., the recursive case). Each clause in this style of S definition is separated by a `|`. The functions `HD` and `TL` are standard library functions for taking the head (i.e., the first element) of a list and the tail (i.e., the rest) of a list respectively.

Type expressions of the form, `:ty1 -> ty2`, are used in the declaration of parameters that are functions from elements of type `ty1` to elements of type `ty2`. Type expressions of the form `:ty1 # ty2` describe tuples. Similarly, type expressions of the form, `:(ty) list`, indicate when a parameter is a list of elements of type `ty`.

Lambda expressions are expressed in S notation as, `\x.E` (where E is an expression).

The semantic definitions for the tabular notation given in the S notation are shown below.

```
(:ty) EveryAux (NIL) (p:ty->bool) := T |
      EveryAux (CONS e tl) p := (p e) and EveryAux tl p;

(:ty) Every (p:ty->bool) l := EveryAux l p;

(:ty) ExistsAux (NIL) (p:ty->bool) := F |
      ExistsAux (CONS e tl) p := (p e) or ExistsAux tl p;

(:ty) Exists (p:ty->bool) l := ExistsAux l p;

(:ty) UNKNOWN : ty;
(:ty)DC := \(x:ty).T;
TRUE := \x. x = T;
FALSE := \x. x = F;

(:ty1) RowAux (CONS (p:ty1->bool) tl) label :=
              CONS (p label) (RowAux tl label) |
       RowAux (NIL) label := NIL;

(:ty) Row label (plist:(A->bool)list) := RowAux plist label;

Columns t := if ((HD t)=NIL) then NIL
             else CONS (Every (HD) t) (Columns (Map t (TL)));

(:ty) TableSemAux (NIL) (retVals:(ty)list) :=
        if (retVals=NIL) then UNKNOWN else (HD retVals) |
      TableSemAux (CONS col colList) retVals :=
        if  col then (HD retVals) else TableSemAux colList (TL retVals);

(:ty) Table t (retVals:(ty)list) := TableSemAux (Columns t) retVals;
```