Using a Mix of Languages in Formal Methods: The PET System

Elsa L. Gunter
Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974
elsa@research.bell-labs.com

Doron A. Peled
Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974
doron@research.bell-labs.com

ABSTRACT

The PET project is a formal methods tool for determining necessary conditions for the execution of a path through a concurrent program. The user selects a path by selecting nodes in flow graphs displayed by the graphical user interface. The condition necessary for that path to be executable is then calculated and displayed for the user. This project mixes a graphical user interface with a back-end for the symbolic computation of the path conditions. The graphical user interface is written in TCL/TK and the back-end is written in SML. TCL/TK is a language specifically designed for the rapid construction of graphical user interfaces. SML is higher-order applicative programming language that is very well suited to symbolic computation. Interprocess communication is handled through pipelining strings. Both TCL/TK and SML are fairly good at string manipulation, so the overhead is tolerable. The ease and efficiency of writing each part in a language best suited for each task more than compensates for this additional overhead.

1. INTRODUCTION

The Path Exploration Tool (PET [5]) is a formal methods tool for determining necessary conditions for the execution of a path through a concurrent program. The user gives an input program in the form of a collection of concurrent processes. Each of these processes is displayed as a flow graph by the graphical user interface. The user selects a path by selecting nodes in flow graphs. The condition necessary for that path to be executable is then calculated and displayed. This project mixes a graphical user interface written in TCL/TK [10] with a back-end for the symbolic computation of the path conditions written in SML [8].

Formal methods are a collection of techniques aimed at increasing the reliability of software and hardware systems. Since the systems under investigation are often quite complex, efficiency is often a main concern for formal methods tools, particularly ones like model checkers. Therefore, the tendency is to develop formal methods tools in programming languages like C, where one can obtain lower level control, even up to the level of forcing some computation to be performed within the machine registers. With the advancement of modern compilers, higher level programming languages can now offer reasonable efficiency in addition to special features not present in lower level languages. These languages allow us satisfy another major concern of formal methods tools, namely that the code be clear and concise, improving the chances that it is correct.

In this paper, we concentrate on a particular strength of functional programming languages, and in particular SML. Specifically, we focus on the ability to perform symbolic manipulations. This capability, standard in functional languages, and in particular in SML, can require a lot of special infrastructure to be prepared in order to be used with a declarative language like C. It allows the calculation of conditions related to certain executions of the checked software, and the verification and testing of software.

In a previous project, we implemented in SML a prototype for translating specifications written in linear temporal logic [11] into a form that allows automatic verification (model checking) [1] of software. Like the back-end of PET, this project was mainly one of symbolic manipulation. This project, reported in [3], resulted in a prototype for this translation, consisting of about 200 lines of SML code, written in a single day of work. Subsequently, the actual implementation of the algorithm that was integrated into the Spin model checking system [6], was propagated over 4 months, and included about 5,000 lines of C code. The main reason for the reimplementation in C was that the Spin system was already written in that programming language.

Equipped with this previous experience, we decided to use SML/NJ as the implementation language for the back-end of our new Path Exploration Tool. This tool is intended for interactive testing of sequential and concurrent software. The main calculation performed by PET is the symbolic manipulation of conditions for executing sequences of program instructions (namely, path conditions).

The front-end, responsible for the graphical interface, is implemented using the graphical interface programming language TCL/TK. The interface between the two parts is done

through a communication protocol, and using files prepared by the back-end part for use by the front-end. In this way, there is only minimal work left for the TCL/TK front-end (which is much slower than SML/NJ), besides the imperative graphical part.

In this paper we describe the PET tool, the decisions that led us to use the functional language SML/NJ and the imperative language TCL/TK for implementing PET, and the interface between the two parts of our project.

2. THE PET SYSTEM

The PET system is an interactive testing [9] tool based on program verification theory. The input to PET, i.e., the tested program, is written in a derivative of Pascal. The syntax for this programming language includes traditional imperative constructs such as *if-then-else*, *while-loop*, *begin-end* block and *assignments*. It is enhanced with constructs from concurrent programming, such as (asynchronous) message passing, *wait* statement, and *random* (nondeterministic) choice.

PET converts the input program (consisting of one or more processes) into a *flow graph*, which is a visual representation of a program. A node in a flow graph is one of the following: *begin, end, predicate, random, wait, assign, send* or *receive*. The *begin* and *end* nodes appear as ovals, the *predicate*, *wait* and *random* nodes appear as diamonds, labeled by a condition, or the word *random*, in the latter case. All other nodes appear as boxes labeled by the *assign*, *send* or *receive* statement.

The main focus of PET is *execution paths* (or *paths* for short). A path is a consecutive sequence of statements from one process, or an interleaved sequence of statements from different processes. PET allows the user to interactively select a path, and modify it by appending or removing statements or changing the relative order between statements from different processes. PET calculates the condition for executing the path and tries to simplify the path condition into a form that will be understood by the user. The path condition is given in first order logic.

A condition is calculated backwards, starting with *true*. Thus, we proceed from a *postcondition* of a transition, i.e., a condition that holds immediately after executing it, to calculate its *precondition*, i.e., the condition that holds immediately prior to executing it. In order to calculate the precondition given the transition and the postcondition, we apply various transformations to the current condition c, depending on the type of transition. For example, for a transition that consists of a predicate p with the 'yes' out-edge, we obtain the precondition $c \wedge p$. The same predicate with the 'no' out-edge, results in the precondition $c \wedge \neg p$. For an assignment of the form $x := e$, we replace every (free) occurrence of the variable x in the postcondition c by the expression e. We start the calculation with the postcondition *true* at the end of the selected path, and proceed backwards to the beginning of the path to obtain the path condition.

The architecture of PET is described in Figure 1. The graphical user interface is created as a TCL/TK program.

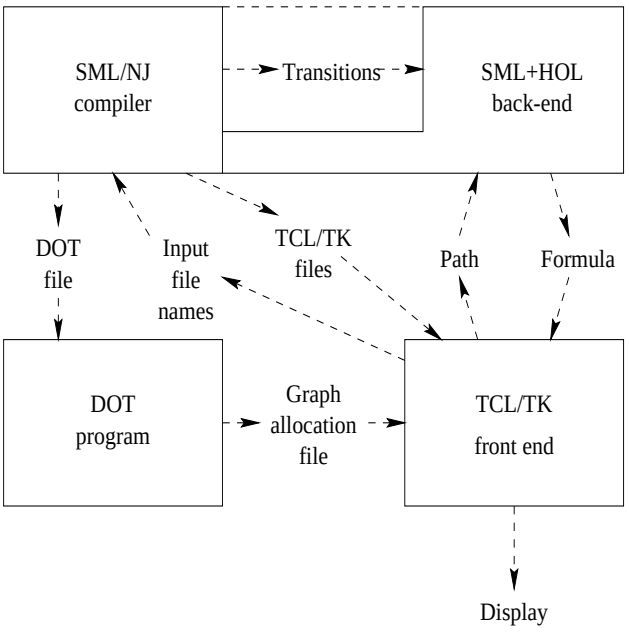

Figure 1: The PET architecture

The back-end symbolic computation is done in SML/NJ. The SML/NJ program contains a compiler, for translating the input Pascal code into internal form. The internal representation of the code is as abstract syntax transitions. The set of abstract syntax transitions corresponds to the flow graph displayed by the TCL/TK front-end. The SML program calculates from this set a listing of the edge relations in the flow graph to be used by the TCL/TK front-end, and a DOT [2] input file containing the components of the flow graph. The SML/NJ program then calls DOT to generate a file containing the layout information used by the TCL/TK front-end to display the flow graph. While parsing the input Pascal code, the SML/NJ back-end also generates tagging information relating nodes in the flow graph back to text in the original code.

The main task performed by the SML back-end is the calculation of path conditions and the simplification of those conditions to make them more readable for the human user. It is here that the strengths of SML for symbolic computation are most important. In addition to performing some simplifications directly, the SML back-end uses simplification routines for Presburger arithmetic from the HOL90 [4] theorem prover, also written in SML, as dynamically linked libraries.

The communication between the front-end and the back-end is done using a two-way pipeline, where each message sent from the front-end to the back-end includes either the name of a file to be compiled or a path for which the path condition is requested. The SML back-end performs the required task, and provides some response. If a compilation is required, some external files, as described above, are created. We extensively use the feature of TCL/TK to import a new file during run-time. This arrangement allows us to transfer most of the computation from the typically slow TCL/TK to the SML back-end.

3. THE TCL/TK GRAPHICAL INTERFACE

Recently, it has become increasingly evident that formal methods tools that are based on a graphical interface achieve a relatively high degree of success. The PET system makes an extensive use of the visual notation of flow graphs. The tested execution paths, which are the main object of PET, are collected and edited by the user using mouse clicking. The output of PET, a path condition written in first order logic, is supplied by the system at a special window. For a sample view of the flow graphs displayed by the graphical user interface see Figure 2.

TCL/TK is an interpreted programming languages for building visual applications. Its syntax is quite simple (although, due to some various modes of enforcing and suppressing interpretation, it may be sometimes confusing to use in the beginning). It allows many visual features, such as a graphical canvas, a textual pad and a large number of menus and buttons. The graphical canvas is used in PET for displaying the flow graph. It allows positioning various shapes with different colors, sizes and line types. The actual location of the flow graph nodes is calculated by the Unix command DOT.

Each shape, menu, button or text unit is organized in TCL/TK as an *object*. Each cursor move is an *event* (e.g., moving the cursor into or out of the boundary of the object). In order to make clearer the connection between the code, the flow graph and the selected path, sensitive highlighting is used. For example, when the cursor points at some predicate node in the flow graph window, the corresponding text is highlighted in the process window. The code corresponding to a predicate node can be, e.g., the relevant *if-then-else* or a *while* loop. This is done by *tagging* objects. For example, a *while-loop* can be tagged by a property that is connected to the move of the cursor inside the node with the loop's predicate. This declarative connection will cause the TCL/TK interpreter to highlight the loop's text when the cursor is inside the loop. A similar arrangement will cause the highlighting to be canceled when the cursor exists that node.

SML/NJ has a library, SML/TK [7], that provides the functionality of TCL. It communicates via pipes directly with TK. This allowed us to rewrite the graphical user interface also in SML, thereby eliminating the need for pipeline communication between the front-end and the back-end.. However, the pipeline communication between SML and TK was itself too slow. An implementation of SML/TK with direct system calls should circumvent that, and is in the planning by the implementers of SML/TK.

4. THE SML BACK-END PROGRAM

The computation engine of the PET system is an SML/NJ program. This program is responsible for performing the following activities:

- Compile the input program and extract from it information in several formats (see Figure 1):
  - An internal representation, where the program statements are represented as *abstract syntax transitions*. Each program construct translates to a description of the transformation to be performed by executing the construct, and a set of transitions labeled with enabling conditions. Effectively, these are the internal representation of the nodes in the flow graph together with there out-edges.
  - A graph, in DOT [2] format. This representation allows using the DOT program in order to generate the flow graph graphical representation.
  - Several files containing tabular information in TCL/TK format. This information allows the graphical interface to perform various tasks such as obtaining sensitive linking between the textual and the graphical representation of the program.
- Perform the calculation of the path conditions.
- Perform the simplification of the path conditions, in order to provide the user with an easy to read formula.
- Communicate with the front-end graphical interface using a two way protocol, and provide calculation and compilations according to need.

The compiler component of the system was constructed using the SML/NJ compiler construction tools ML-Lex and ML-Yacc (which are similar to the C tools with corresponding names). The compilation process results in a representation of the program as a set of nodes with out-edges. The built-in recursive type declaration of SML/NJ supports this representation directly, unlike imperative programming languages.

This kind of computation is very natural for a functional programming language like SML. It allows the definition of recursive types that can represent first order logic terms and formulas:

```
datatype term =
   C of int | V of var_name
 | neg of term | plus of term*term
 | times of term*term | minus of term*term
 | dvd of term*term | pow of term*term
 | condition of pred*term*term

 term_pred =
   gr of term*term | lt of term*term
 | ge of term*term | le of term*term
 | eq of term*term | ne of term*term

and pred =
   True | False | Term_pred of term_pred
 | And of pred*pred | Not of pred
 | Or  of pred*pred | Implies of pred*pred
```

To demonstrate the natural way in which this calculation is performed in SML/NJ, consider the following excerpt, which substitutes a variable (the first parameter) by a term (the second parameter) in another term (the third parameter).

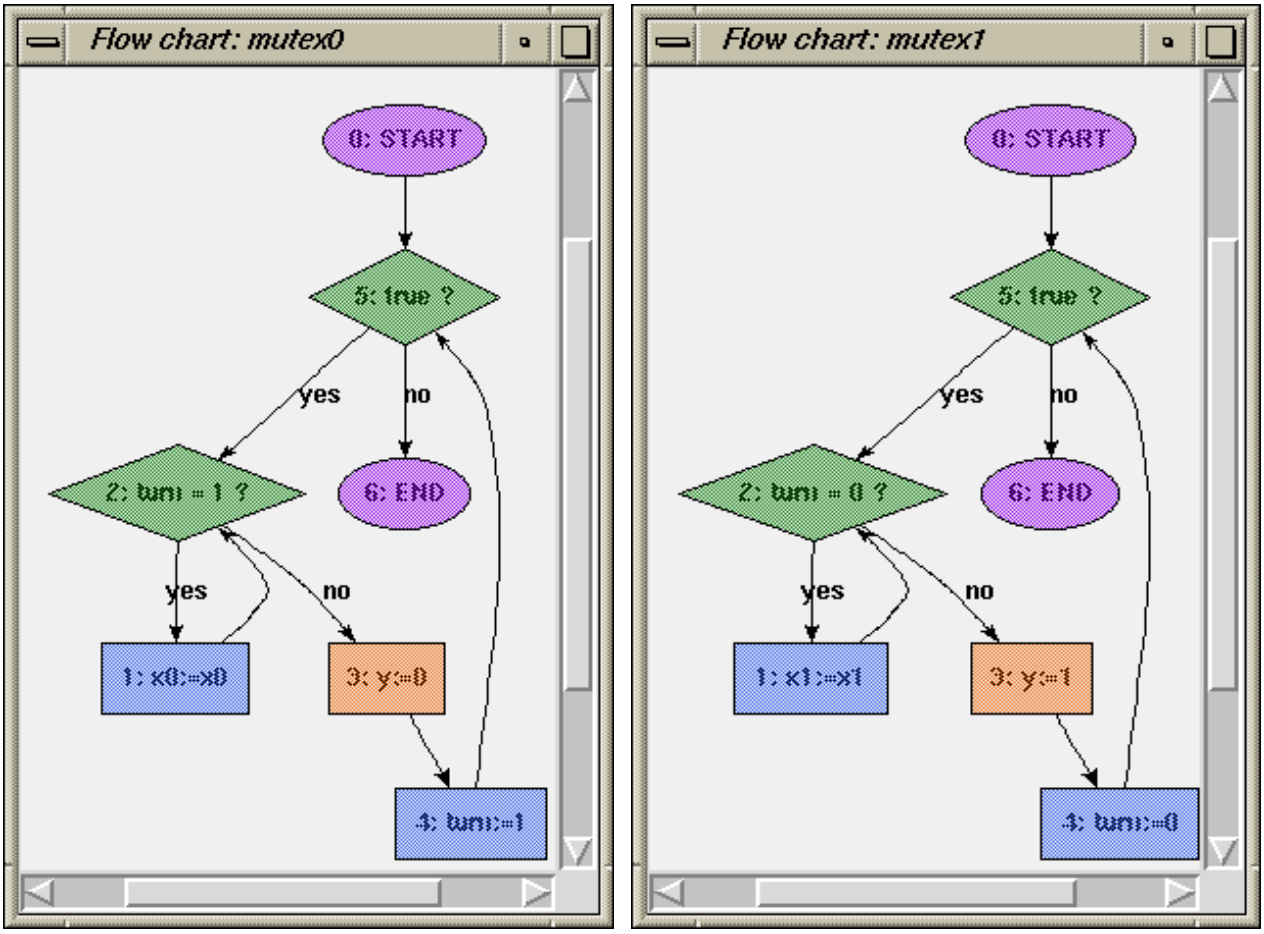

Figure 2: A Sample View of Flow Graphs

```
fun subst_term(_, _, C(x)) = C(x)
  | subst_term(x, t, V(y)) =
     if x=y then t else V(y)
  | subst_term(x, t, neg y) =
     neg(subst_term(x,t,y))
  | subst_term(x, t, plus(y, z)) =
     plus(subst_term(x, t, y),
          subst_term(x, t, z))
...
  | subst_term(x, t, condition(p, y, z)) =
     condition(subst_pred(x, t, p),
               subst_term(x, t, y),
               subst_term(x, t, z))

and subst_term_pred(VAR, TERM, gr(x,y))=
     gr(subst_term(VAR, TERM, x),
        subst_term(VAR, TERM, y))
  | subst_term_pred(VAR, TERM, lt(x,y))=
     lt(subst_term(VAR, TERM, x),
        subst_term(VAR, TERM, y))
...

and subst_pred(_, _, True)=True
  | subst_pred(_, _, False)=False
  | subst_pred(VAR, TERM, And(x,y))=
     And(subst_pred(VAR, TERM, x),
         subst_pred(VAR, TERM, y))
...
  | subst_pred(VAR, TERM, Term_pred( x )) =
     Term_pred(subst_term_pred(VAR, TERM, x))
```

The user selects a path in the flow graph by clicking on a sequence of edges. However, for the computation of the path condition, it is not really the sequence of nodes that matters, but rather the sequence of transitions between the nodes that is important. It's not good enough to know that we are at a conditional node; we need to know whether we follow the 'yes' transition or the 'no' transition. Therefore, we must translate the sequence of nodes selected by the user into a corresponding list of transitions. The transitions are represented using the following datatype:

```
datatype arc =
    begin of ...
  | asmt of {var: var, asg_value:term, ...}
  | yes_cond of {cond:pred, ...}
  | no_cond of {cond:pred, ...}
  | undetermined_cond of ...
  | skip of ...
  | terminal of ...
```

We have abbreviated the above datatype somewhat, omitting some arguments not directly needed for the computation of the path condition.

Once we have a list of transitions, the calculation of the path condition is now quite straight-forward:

```
fun one_step_wpc {post_cond,
                  arc = asmt{var, asg_value,...}} =
    subst_pred(var,asg_value,post_cond)
  | one_step_wpc {post_cond,
                  arc = yes_cond{cond,...}} =
    And(cond,post_cond)

  | one_step_wpc {post_cond,
                  arc = no_cond{cond,...}} =
    And(Not cond, post_cond)
  | one_step_wpc {post_cond, ...} = post_cond

fun wpc {post_cond, path} =
    List.foldl
    (fn (arc,pred) =>
     simplify(one_step_wpc{post_cond = pred,
                           arc = arc}))
    True path
```

The simplification of a formula in PET is based on simple rewriting rules. Additional simplification is performed by using a simplification procedure that was implemented as part of HOL. This simplification procedure can decide whether a subformula in Presburger arithmetic is a tautology.

The SML back-end is responsible for receiving commands from the TCL/TK front-end and acting upon them. The TCL/TK front-end communicates to the SML back-end by sending strings to the standard-in of the SML back-end program. The language the front-end speaks to the SML back-end is a simple language, which requires only lexing from ML_Lex, and simple pattern matching to parse.

The SML back-end responds to TCL/TK similarly by sending strings, and additionally by writing various information to files. The response language requires even less analysis than the input language. String pattern matching in TCL/TK is sufficient to handle this. This response language tells when the back-end in ready for input, when it has finished writing information to files, flags if an error has occurred, and gives strings stating the simplified path condition for any path chosen so far.

As we have developed PET, we had extended it functionality in several components. We have experimented with the addition of new language constructs and the creation of various forms of abstractions of the input programs. These extension have been done with a minimum of code rewriting. For the addition of new language constructs, the recursive datatypes themselves sometimes required additional clauses be added, and then the corresponding clauses needed to be added to the functions over them. We have yet to have an extension that required us to rewrite the exiting clauses remaining from the original types. Other extensions have been accomplished by adding new code modules (SML "structures") and modifying only the code in the top-level function that interfaces the back-end to the TCL/TK front-end. Particularly in the process of experimenting with various extensions, we found it very useful to make liberal use of SML/NJ library for data structures like hash tables and FIFO queues. The TCL/TK interface typically required no rewriting for the addition of new language constructs, but naturally it required the addition of procedures when the extensions required new display windows.

5. CONCLUSIONS

The most important feature of SML/NJ for its use in building PET was its support for user-defined recursive datatypes, and function definition by pattern matching over these datatypes. This allowed us to write code quickly, concisely, and in a manner that could readily be updated as new features were tested out in the system. Beyond this, some of the features that made SML/NJ a good choice for this project include ML-Lex, ML-Yacc, libraries for data-structures like hash-tables, the ability to call external procedures like DOT, and the ability to call procedures in HOL90 as a library. The module system has proved of particularly great value as we have experimented with various extensions of the system. The lack of rich and efficient support for the development of graphical user interfaces made SML/NJ unsuitable for the entire project, but is ability to cooperate with a TCL/TK front-end allowed us to overcome this short-coming.

Functional programming languages supporting user-defined recursive datatypes, and function definition by pattern matching can be very efficient for the implementation of formal methods tools requiring large amounts of symbolic manipulation. To be truly useful, other features such as a rich library providing programming infrastructure, and a module system to support incremental development are also needed. Functional programming languages will not always be the best tool for every application, if for no other reason than that the library structure supporting an application may not yet have been developed for the functional language. Therefore, it is also necessary for functional programming languages to have strong support for interprocess communication. In building the PET tool we were able to make due with communicating through strings passed between the processes. However, we were unable to make use of the SML/TK library because fundamentally pipeline communication was too slow for human interaction. With better interprocess communication, the SML/TK library would probably be able to overcome this barrier.

6. REFERENCES

[1] E.M. Clarke, E.A. Emerson, Design and synthesis of synchronization skeletons using branching time temporal logic. Workshop on Logic of Programs, Yorktown Heights, NY, 1981, LNCS 131, Springer-Verlag.

[2] E.R. Gansner, S.C. North, K.P. Vo, DAG - A Program to Draw Directed Graphs, Software – Practice and Experience 17(1), 1988, pp. 1047-1062.

[3] R. Gerth, D. Peled, M. Vardi, P. Wolper, Simple on-the-fly automatic verification of linear temporal logic, Protocol Specification Testing and Verification, 1995, Chapman & Hall, 3–18, Warsaw, Poland.

[4] Michael J. C. Gordon and Thomas Melham. *Introduction to HOL*. Cambridge University Press, 1993.

[5] E.L. Gunter, D. Peled, Path Exploration Tool, to appear in Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Amsterdam, 1999.

[6] G. J. Holzmann, The model checker SPIN, IEEE transactions on Software Engineering, 23(5):279–295.

[7] C. Lüth, B. Wolff, Functional Design and Implementation of Graphical User Interfaces for Theorem Provers. To appear in the Journal of Functional Programming.

[8] R. Milner, R. Harper, D. B. MacQueen. *The Definition of Standard ML*. MIT Press, 1997.

[9] G.J. Myers. *The Art of Software Testing*. John Wiley and Sons, 1979.

[10] J. K. Ousterhout. *Tcl and Tk toolkit.* Addison-Wesley, 1994

[11] A. Pnueli, The temporal logic of programs, *18th FOCS, IEEE Symposium on Foundation of Computer Science*, 1977, 46–57.