Efficient Symbolic Simulation of Low Level Software

Tamarah Arons, Elad Elster, Shlomit Ozer, Jonathan Shalev and Eli Singerman
Intel Israel Design Center
E-mail: firstname.lastname@intel.com

Abstract

Symbolic execution has long been a staple technique for formal hardware verification. Its application to software requires methods for dealing with software specific complexities. In this paper we elaborate methods for the efficient symbolic simulation of embedded software; some methods are new, others are improvements of existing methods. Using these techniques we have been able to symbolically execute real life microcode of thousands of lines, allowing formal methods to become an integral part of microcode validation in Intel Corporation.

```
normalize: (REG2≥0)?  goto reduce;
in_norm:   REG2 := -REG2;
reduce:    (REG2≤10)?  goto e_norm;
in_red:    REG2 := 10;
e_norm:    goto REG1;

start:     REG1 := e_flow;
           (EAX < EBX)? goto eax_s;
           REG2 := EBX;
           goto normalize;
eax_s:     REG2 := EAX;
           goto normalize;
e_flow:    EAX := REG2;
```

Figure 1. Example program.

1 Introduction

Symbolic execution, which has been successfully used in the formal verification of hardware [4], is now also being applied to software of various levels [6]. Such software may be a higher level abstraction of RTL [9], more general C programs [3], or various types of low-level software [7]. We have developed the MICROFORMAL tool-suite, originally used in the formal verification and analysis of microcode, and currently being extended for the verification of other embedded software. The MICROFORMAL tool-suite supports formal property verification, formal equivalence verification [1], and the extraction of feasible paths to serve as a coverage metric [2]. In this paper we focus not on the individual applications, but on their shared underlying symbolic simulation technology.

In order to symbolically execute real microcode programs, some of which are thousands of lines long, with broad branching, we developed a variety of methods for dealing with complexity in software simulation. Our basic theory remains that of [1], but our methodology has, of necessity, become significantly more sophisticated.

One significant contribution is a theory of *configurable merge points* suitable for large programs. Merge points (Sec. 3) unify simulation branches, allowing the simulation of significantly larger programs. We discuss a system of implicit and explicit merge points, which significantly speeds up simulation. We present novel methods to cache and re-use simulation information, particularly SAT-queries (Sec. 5). We describe an algorithm for the effective resolution of indirect jumps in Sec. 4.

The paper is structured as follows: We first give a brief overview of symbolic simulation. We then detail the aforementioned methods. Finally we give results and compare our methods with those in the literature.

2 Symbolic Simulation

Symbolic simulation of software is similar to that of hardware. A formal definition of our semantics can be found in [1]. Here we give a brief, intuitive, explanation.

In symbolic simulation a program is simulated, but the initial states (variables such as registers) do not have a concrete initial value, but rather a symbolic value. During the course of the simulation these are updated to be symbolic expressions over the initial values and constants that appear in the program. When a conditional jump occurs, we use a SAT solver to determine which of the condition and its negation is satisfiable in the symbolic state. When both are satisfiable we create a new *path* containing a new symbolic state. The symbolic simulation effectively creates a tree of

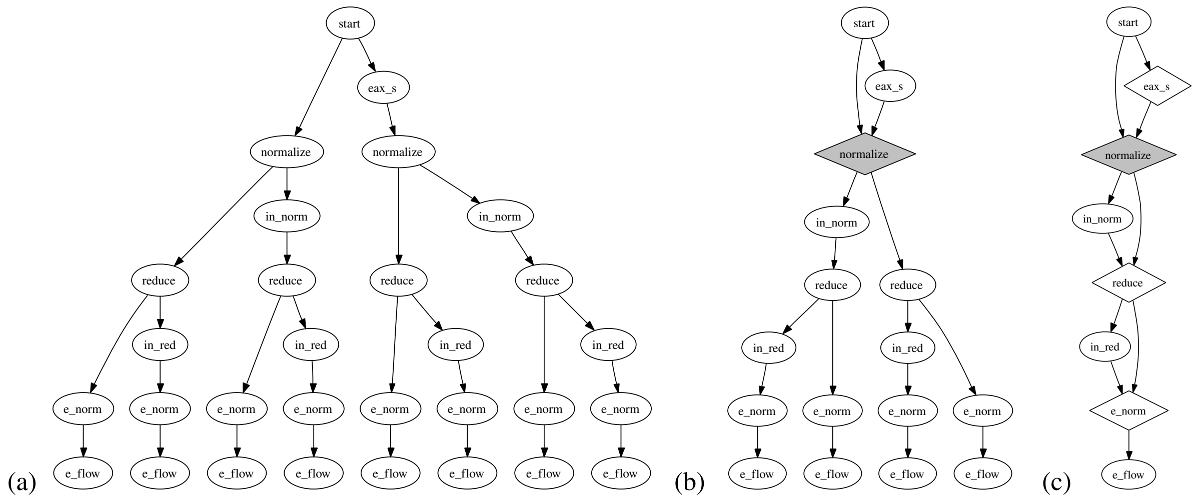

Figure 2. Symbolic simulation of program start. (a) No merge points (b) Explicit merge point at normalize (c) Explicit and implicit merge points. Shaded and plain diamonds denote explicit and implicit merge points, respectively.

simulation paths, which branch at conditional gotos. Each path has a symbolic final state, and a condition under which it is followed.

We demonstrate this on the first five lines of the *normalize* procedure, lines 1–5 of Fig. 1. Procedure *normalize* takes two inputs REG1, the return address, and REG2, a value to be "normalized" to the range $[0, 10]$. REG2 is updated to its absolute value. Any value exceeding ten is replaced by ten.

At the start of the simulation there is single path, with *path condition* TRUE. The jump at *normalize* may or may not be taken, depending on the initial value $\texttt{REG2}_0$ of REG2. We therefore split the path into two, one path with condition $\texttt{REG2}_0 \geq 0$, and another with condition $\texttt{REG2}_0 < 0$. These two paths are each split at *reduce*, so that when *e_norm* is reached, there are four simulation paths.

We turn now to program *start*, at line 6. It normalizes the smaller of EAX and EBX storing the result in EAX. Program start generates two paths, jumping to normalize at lines 9 and 11, respectively. Continuing the simulation through the normalize procedure results in a total of 8 paths – the full normalize subtree, with four paths, is simulated for for each call of normalize. See Fig. 2(a).

The parameter to normalize is passed in REG2 and the return address in REG1, as is often the case in low level software where procedures and call stack mechanisms are not inbuilt. This requires the manipulation of labels within expressions, discussed in Sec. 4.

We use a stack based model of memory similar to that of [9]. We employ expression renaming and sub-expression sharing, similar to that of [8, 3].

Loops and Simulation Tracking Symbolic simulation sometimes takes unexpectedly long. One common reason is that a loop is being simulated. Alternatively, the simulator may have reached a complex subprogram in an unexpected manner. We track simulation progress, allowing the user to understand why simulation is taking a long time, and possibly add assumptions to prune irrelevant branches. This is done by periodically generating graphical control-flow graphs (CFG), similar in format to those of Fig. 2(c). The new paths (relative to the previously outputed CFG) are colored, indicating which part of the program is currently being simulated.

When a loop is identified, MICROFORMAL exits, providing information about the loop and how it was reached. We support a number of unsound methods of reducing the number of loop iterations: The user may specify the maximum number of times a location may be visited. The tool then generates an appropriate constraint on the initial state to ensure this. Alternatively, the user may explicitly overwrite a counter value, or replace a loop with other code.

3 Merge Points

Merge points are explained in depth in [9] and are used by many [6, 7]. Methods which convert if-then-else statements and simple loops into conditional assignments effectively merge variable values by unifying the path simulators [10]. However, such local merges are ineffective in programs with conditional jumps to large sub-procedures. Our semantics is closer to [9], maintaining multiple explicit simulation paths which are merged together at various points.

The simulation tree of Fig. 2(a) includes two duplicate subtrees each rooted at *normalize*. Although *normalize* is called with different parameters, its symbolic simulation is identical whether it is called from line 9 or 11. Simulation can be sped up by simulating it once only. To do this we use *merge points*, locations at which multiple simulation paths are merged into a single simulation path. The merge of simulation paths at location l is defined as follows:

Let P be a set $p_1, p_2, \ldots, p_n$ of simulation paths waiting at location l with path conditions $c_1, \ldots, c_n$ and final states $s_1, \ldots, s_n$, respectively. Every state s_i is defined as a set of variables $v_1, \ldots, v_m$; we denote the value of variable v_j in state s by $s.v_j$. The merged condition C, and merged state S, of P are defined as:

$$\begin{aligned} C &= \bigvee_{i=1}^{n} c_i \\ S &= \{v_j : j = 1 \ldots m\} \text{ where} \end{aligned}$$

$$v_j = \begin{cases} \text{IF} & c_1 & \text{THEN} & s_1.v_j; \\ & & \cdots & \\ \text{IF} & c_{n-1} & \text{THEN} & s_{n-1}.v_j; \\ & & \text{ELSE} & s_n.v_j \end{cases}$$

Both the c_i and the v_j are typically symbolic (rather than constant) expressions.

Returning to our example of Fig. 1, we define a merge point at location *normalize*. We start simulation at instruction 6 (*start*) with initial state s_0 = $\{\texttt{REG1=}\ \texttt{REG1}_0, \texttt{REG2=}\ \texttt{REG2}_0, \texttt{EAX=}\ \texttt{EAX}_0, \texttt{EBX=}\ \texttt{EBX}_0\}$ and path condition TRUE. Two paths, p_1 with condition $(\texttt{EAX}_0 < \texttt{EBX}_0)$, and p_2 with condition $!(\texttt{EAX}_0 < \texttt{EBX}_0)$ reach *normalize*. The merge state at *normalize* is built as

$$C = (\texttt{EAX}_0 < \texttt{EBX}_0) \vee !(\texttt{EAX}_0 < \texttt{EBX}_0)$$

$$S = \left\{ \begin{array}{l} \texttt{EAX} = \texttt{EAX}_0;\ \texttt{EBX} = \texttt{EBX}_0; \\ \texttt{REG1} = \textit{e\_flow}; \\ \texttt{REG2} = (\texttt{EAX}_0 < \texttt{EBX}_0)\ ?\ \texttt{EAX}_0 : \texttt{EBX}_0 \end{array} \right\}$$

We note that in this case C can be simplified to TRUE, as all the paths reach the merge point. However, typically only a subset of the simulation paths reach a merge point.

The simulation from the merge point continues normally, with some variables being represented as conditional expressions over the variables of paths reaching the merge point. The simulation (Fig. 2(b)) takes roughly half as long as the original (Fig. 2(a)), with half as many steps being simulated.

3.1 Configurable Merge Points

Existing merge point algorithms use rigid definitions of where merges should occur. Typically, every location is a merge point. In [9] this is refined by giving different priorities to merge points, based on fan-in. We deal with code which is thousands of lines long, with thousands of simulation paths. In our case it is not effective to merge at every location, nor is fan-in priority sufficient. We developed a 2-tiered system of configurable merge points.

```
l1 :  c?  goto l2;          l1 :  c?  goto l2;
            ...                         ...
l2    ...                   l3 :  complex code
            ...                         ...
l3 :  complex code          l2    ...
            ...                   goto l3;
```

Figure 3. (a) l_2 precedes l_3 (b) l_3 precedes l_2

The user defines sets of *implicit merge points* by the criteria which they fulfill. A frequently used, and effective, heuristic is that the target of any direct jump (conditional or not) is an implicit merge point. Using this heuristic, the locations *normalize*, *reduce*, *e_norm* and *eax_s* are the implicit merge points of program *start*. The user does not list the relevant merge points, but rather updates the formal definitions (templates [1]) of the relevant operations. This method is therefore very scalable, requiring little user intervention.

In addition, the user may define a set of *explicit merge points*. Each of these merge points is defined explicitly, by label. Most explicit merge points also qualify as implicit merge points; their benefit is *synchronization*. The merges at explicit merge points are delayed until no further implicit merges can be performed. While implicit merge points may synchronize between paths which are very similar, their multiplicity frequently allows one branch to progress faster than another. The rarer explicit merge points function as synchronization points for these diverged simulations, with all paths waiting for others to catch up.

We demonstrate the benefit of synchronization points on the code snippet of Fig. 3. Assume l_1, l_2 and l_3 are all implicit merge points, and that the code following l_3 is complex and for performance reasons should be simulated once only. If l_2 lies between l_1 and l_3 (Fig. 3(a)), then merging the simulation points in the order in which they occur in the code will give adequate performance. However, if l_2 is a procedure placed after l_3 (Fig. 3(b)), then code order will result in l_3 being merged before l_2 and the complex computation is simulated twice. The simulator cannot predict at l_1 that the two paths will converge at l_3 (rather than l_2) and no heuristic will always give the correct simulation order. The user can provide this information about the program structure by defining l_3 as an explicit merge point. This synchronizes all the simulation paths at this point, guaranteeing efficient performance.

Explicit merge points should be placed at the beginning of what is, semantically, a new algorithm or significant procedure in the program. They are also sometimes placed just before loops or other computationally intensive pieces of code in order to ensure that that code is simulated only once. In practice, most explicit merge points are identified and added when code simulation for a specific program takes too long. The graphical control-flow graph generated of the

```
Let $x_1, \ldots, x_n$ be the set of explicit merge points
Let $m_1, \ldots, m_p$ be the set of implicit merge points

Procedure simulate_with_merges()
    Simulate all open paths
    For $i = 1 \ldots$ MAX_ITERATIONS
        For $j = 1 \ldots n$
            do_implicit_merges()
            If there are no open paths, terminate
            If only one path reaches $x_j$
              change its status to open
            If two or more paths reach $x_j$
              merge the paths into a new open path
            Simulate all open paths
    If any open paths remain, report a potential loop

Procedure do_implicit_merges()
    For $i = 1 \ldots$ MAX_IMPLICIT_ITERATIONS
        For $j = 1 \ldots p$
            If only one path reaches $m_j$
              change its status to open
            If two or more paths reach $m_j$
              merge the paths into a new open path
            Simulate all open paths
```

Figure 4. Merging algorithm

parts which have been simulated (Sec. 2) is used to manually identify potential merge points – often those with a high fan-in, or known "signposts" in the expected execution. It is typically easy for designers or validators viewing the CFG to define labels for explicit merge points, as these are the salient program stages with which they are familiar.

A simplified version of our algorithm is presented in Fig. 4. A path is called *terminated* when it has reached the end of the code. It is *waiting* if it is has reached a merge point, and is called *open* if it is neither terminated nor waiting. An open path can always be simulated. Since paths are split at jumps, there may be many open paths at one time.

The effectiveness of a merge point is proportional to the number of paths leading to it. Explicit merges have the highest importance and their merges are therefore delayed as long as possible to allow more incoming paths to be collected. This also gives their synchronization effect. The less important implicit merges are performed first.

MICROFORMAL simulates all open paths until they either terminate or reach a merge point. When no open paths remain, the next (in order of line number) implicit merge point is chosen. If no paths are waiting at this point, nothing is done. If only one path is waiting, the path simulation is simply continued, without a merge being performed. If two or more paths reach the merge point, they are merged and the resultant state is simulated until all its paths terminate or reach merge points. Once no more implicit merges can be done, an explicit merge point is merged. This process continues until either all paths are terminated, or a maximal number of iterations have been completed. The latter case generally indicates a loop in the simulated code – simulation is halted, and the user is given a trace of the path leading to the loop.

The handling of implicit and explicit merge points is symmetric, and our merging algorithm could be parameterized to an arbitrary number of priority levels if necessary.

We now demonstrate merge points on our example. We assume that *normalize*, being that start of a procedure, has been identified as an explicit merge point. The program is simulated until one path reaches *eax_s* and the other *normalize*. Location *eax_s* is an implicit merge point, i.e. of less importance, and so it is considered first. Since only one path reaches *eax_s*, a merge does not take place, instead the path continues simulating to *normalize*. At this point all paths are at explicit merge points, and the merge at *normalize* is performed. We then continue simulating, merging two paths at each of the implicit merge points *reduce* and *e_norm*. This simulation is shown in Fig. 2(c).

In this example every statement is simulated once only. This is an ideal situation, and is not typically the case. Had *normalize* not been defined as an explicit merge point, it would have been merged before *eax_s* (as it occurs before *eax_s* in the code) and so all of procedure *normalize* would have been simulated twice.

3.2 Effectiveness of Merge Points

Fig. 5 demonstrates the effectiveness of merge points on various flows. The number of paths indicates the degree of branching in the symbolic simulation (without merge points). The longest path is the number of statements simulated on the longest simulation path. These are not instructions of the original program but rather statements in the our compiled program which is significantly longer [1]. Each of these statements is comparable to a simple C statement.

We note that for some programs implicit merge points suffice. This is true particularly of flows which are linear, with few jumps, or which are particularly small. There are even cases where the overhead of explicit merges slightly outweighs their benefit (P3). There are other programs where implicit merge points provide relatively little benefit (P4). For larger programs (P1, P2) a combination of both implicit and explicit merge points is best.

4 Indirect Jumps

An *indirect jump* is a jump to a target expression, rather than to a simple label. These are particularly problematic

Program	Number of paths	Longest path	Number of explicit merge points	Simulation time (seconds) by type of merge points			
				None	Implicit	Explicit	Implicit and explicit
P1	3040	80645	5	45000+	23050	5040	2816
P2	2784	39709	1	19769	547	1696	286
P3	767	20932	3	10357	2188	6252	2206
P4	130	20606	3	1790	1643	918	888

Figure 5. The effect of implicit and explicit merge points on simulation times

for symbolic simulation, as the simulator does not have a clean set of labels to jump to. Instead, it must analyze an expression, and deduce what the possible jump targets may be. The expression complexity is aggravated by the presence of merge points, which combine the variable values from different paths. The result is frequently a complex expression including both labels and computational / control expressions.

Like [6] we treat control values (labels in a program) as a special type during simulation. However, unlike [6] we allow their manipulation within complex expressions; most of our standard operations including shift, mask, sub-vector extraction, if-then-else and logical operators can be applied to registers containing control values.

A depth first search of the target expression is performed, and all labels are extracted as potential targets. For each potential target we a build a predicate asserting that the target expression resolves to this potential target, given the path condition at this point.

A SAT solver is used to evaluate the feasibility of each of these potential targets. A successful resolution of the indirect jump is reached if a set of feasible targets is found that covers all initial states leading to the indirect jump. The completeness of the set is checked using the SAT solver.

The jump target condition may be very large, and include a number of both viable and non-viable targets. For example, assume we jump to expression $e = [(c)?$ *invalid_target*$:(f(l_1))?l_2 : l_3]$ where c is an arbitrary conditions which does not contain labels, *invalid_target* is an expression which neither contains labels, nor resolves to one, and $f(l_1)$ is some expression over l_1. We generate three predicates, $e = l_1, e = l_2, e = l_3$, of which only the last two are possibly satisfiable if l_1 is distinct from l_2, l_3. We also check completeness: $(e = l_2 \vee e = l_3) = e$. The solution set is complete only if c is unsatisfiable as *invalid_target* is not a viable target.

5 Caching SAT Solver Calls and Jump Targets

A SAT solver is used to check the viability of potential jumps (Sec. 2) and the feasibility of jump targets (Sec. 4).

Program	Simulation time (seconds)		Percentage hits
	No caching	Caching	in SAT cache
P1	4020	2816	40
P2	1621	286	85
P3	2634	2206	16
P4	933	888	18

Figure 6. The effect of caching

As the number of path simulators increases, the size and complexity of the path conditions also increases, and calls to the SAT solver, which always include the path condition, take a greater portion of the simulation time. In long flows, over half the time can be spent in calls to the SAT solver.

In order to reduce the time spent on SAT solving, we cache SAT query results. The same expression is sometimes sent repeatedly to the SAT solver. Frequently we check an expression of which a sub-expression has already been queried; use of the cached values sometimes allows us to conclusively evaluate the satisfiability of the larger expression. For example, if we already found that a is unsatisfiable, then we need not call the SAT solver to check the satisfiability of $a \wedge b$. Our data structure identifies syntactically identical expressions as such, as well as some equivalent expressions ($a \wedge b$ and $b \wedge a$ will share the same expression, due to operand re-ordering).

In theory, some of the benefits from caching could have been obtained by using incremental SAT. We found that doing the caching ourselves was preferable – it could be done at a word, rather than a bit level, saving the need to bit-synthesize expressions which could be solved through the cache. Furthermore, the memory overhead of the caching was extremely low, as the expressions were stored in any event, and only their evaluation had to be added.

Different indirect jumps in the same simulation often reach the same targets. When a jump is analyzed, its feasible targets are cached. In the analysis of subsequent jumps, the feasibility of potential targets is checked in the order of how recently they were placed in the cache. This frequently allows us to check only a subset of the potential targets.

The efficiency of caching on the four programs of Fig. 5

can be seen in Fig. 6. (All simulations in Fig. 5 used caching, we disable caching to measure its effect.) The cache hit rate differs widely during different programs, depending on the degree to which expressions are repeated. In general, the larger and more complex a program is, the more effective caching is.

6 Conclusion

The MICROFORMAL system has been under intensive research (in collaboration with academia) and development at Intel Corporation since 2003. With increased usage the requirement that it operate efficiently and automatically on a wide range of microcode programs grew stronger. The methods listed in this paper were all developed after the tool was already in use, in order to boost performance and increase automation. We compare them with other methods documented in the literature.

The merge algorithm gave MICROFORMAL a very significant boost, allowing us to simulate a number of previously intractable programs. The most similar work to it is that of [9], where all locations are merge-points, and priority between them is determined according to fan-in. In our tests, treating implicit merge points according to fan-in priority as opposed to the priority determined by the line number resulted in a performance degradation in most programs. Fan-in is a good, generic heuristic. However, in our programs, where control flows primarily from top to bottom, code sequence priority reduces path divergence.

The major advantage of our method over this is that the user may *configure* which locations should be merge points and the use of explicit merge points as a *synchronization* mechanism. The programs in [9] are relatively small, with a maximum of 5275 locations simulated. In such programs implicit merge points generally suffice. The benefit of using expert knowledge of program semantics (the identification of explicit merge points) for synchronization is far more significant in large programs (paths of length 35000+) where simulation paths become desynchronized. So, our implicit merge points can be considered a specialization of [9] for the type of software that we simulate, while explicit merge points are an improvement necessitated by larger programs.

Like [3], our verification is both path- and context-sensitive. [3] exploit the natural function-level abstraction boundaries, such as procedure calls, of high-level software like C/C++. Our code has no such clear abstraction boundaries. While merge points can be seen as an attempt to use abstraction boundaries, there is no clear definition of which variables might be modified at any such boundary.

Our jump resolution and label handling is more general than any we have seen in the literature. Like [6], we treat control values (labels in a program) as a special type during simulation. However, unlike [6] we allow their manipulation within complex expressions; most of our standard operations including shift, mask, sub-vector extraction, if-then-else and logical operators can be applied to registers containing control values. When an indirect jump is executed, the target expression must evaluate to a control value for each initial state.

The caching of SAT queries and jump targets gives a very significant speed-up (averaging 35 – 40 percent). While the use of a SAT-solver to prune infeasible paths during symbolic execution is not new [5] we believe that the caching of such values to avoid the querying of related expressions is.

We are continually searching for new methods to get MICROFORMAL to run more efficiently in its various applications. Our current focus is on adapting MICROFORMAL to other types of embedded software and on identifying methods to re-use information from one run in subsequent runs.

References

[1] T. Arons, E. Elster, L. Fix, S. Mador-Haim, M. Mishaeli, J. Shalev, E. Singerman, A. Tiemeyer, M. Y. Vardi, and L. D. Zuck. Formal verification of backward compatibility of microcode. In *Proc. 17^{th} Intl. Conference on Computer Aided Verification (CAV'05)*, pages 185–198, July 2005.

[2] T. Arons, E. Elster, T. Murphy, and E. Singerman. Embedded software validation: Applying formal techniques for coverage and test generation. In *Seventh International Workshop on Microprocessor Test and Verification, 2006. (MTV '06)*, pages 45–51, December 2006.

[3] D. Babic and A. J. Hu. Structural abstraction of software verification conditions. In *Proc. 19th Intl. Conference on Computer Aided Verification (CAV'07)*, pages 366–378, 2007.

[4] R. E. Bryant. Symbolic simulation techniques and applications. In *DAC '90: Proceedings of the 27th ACM/IEEE conference on Design automation*, pages 517–521, New York, NY, USA, 1990. ACM Press.

[5] E. M. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 368–371, 2003.

[6] D. Currie, X. Feng, M. Fujita, A. J. Hu, M. Kwan, and S. Rajan. Embedded software verification using symbolic execution and uninterpreted functions. *Int. J. Parallel Program.*, 34(1):61–91, 2006.

[7] D. W. Currie, A. J. Hu, and S. Rajan. Automatic formal verification of dsp software. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 130–135, 2000.

[8] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *POPL'01*, pages 193–205, 2001.

[9] A. Koelbl and C. Pixley. Constructing efficient formal models from high-level descriptions using symbolic simulation. *Int. J. Parallel Program.*, 33(6):645–666, 2005.

[10] S. Minato. Generation of BDDs from hardware agorithm desciptions. In *Intl. Conf. on Computer-Aided Design (ICCAD'96)*, 1996.