

Memory Modeling in ESL-RTL Equivalence Checking

Alfred Koelbl
Advanced Technology Group
Synopsys, Inc.
2025 NW Cornelius Pass Rd.
Hillsboro, OR 97124
koelbl@synopsys.com

Jerry R. Burch
Advanced Technology Group
Synopsys, Inc.
2025 NW Cornelius Pass Rd.
Hillsboro, OR 97124
burch@synopsys.com

Carl Pixley
Advanced Technology Group
Synopsys, Inc.
2025 NW Cornelius Pass Rd.
Hillsboro, OR 97124
cpixley@synopsys.com

ABSTRACT

When designers create RTL models from a system-level specification, arrays in the system-level model are often implemented as memories in the RTL. Knowing the correspondence between ESL arrays and RTL memories can significantly reduce the complexity of a formal equivalence check between the ESL model and the RTL. In practice, however, handling memory mappings in ESL-RTL equivalence checking is non-trivial for the following reasons: First, because of a lack of bit-accurate data-types in the system-level language, the information stored in an array location may be stored in a compressed form in the RTL. Second, a single array in the ESL model may be implemented by multiple memories in the RTL and/or corresponding data items may be stored in different locations. And last but not least, due to timing differences between the ESL model and the RTL, the correspondence between arrays and memories may not hold in every clock cycle. In this paper, we propose an approach to ESL-RTL equivalence checking which can deal with all of these difficulties.

Categories and Subject Descriptors

J.6 [Computer-Aided Engineering]: Computer-Aided Design

General Terms

Algorithms, Verification

Keywords

ESL, Equivalence checking, Verification, Memory

1. INTRODUCTION

With the rising complexity of modern embedded systems, a growing trend toward designing hardware at higher levels of abstraction has emerged. Many companies have adopted a methodology that starts out with a design specification in a system-level programming language like C++. In this phase, the focus is purely on algorithm design and design exploration. A higher abstraction level enables faster design changes and more thorough validation due

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC 2007, June 4–8, 2007, San Diego, California, USA.

Copyright 2007 ACM 978-1-59593-627-1/07/0006 ...\$5.00.

to a higher simulation speed. By adding more and more implementation details, the system-level specification is then refined to a Register-Transfer-Level (RTL) description.

Today, the prevalent technique for checking if the resulting RTL complies with the system-level specification is simulation. As exhaustive simulation is not feasible for industrial-sized designs, subtle bugs in the design frequently remain undetected. This problem gives rise to the use of formal techniques. Formal equivalence checking can detect discrepancies which are not detected in week long simulation runs.

Whereas formal equivalence checking for RTL to RTL comparisons has been in production use for many years now, ESL to RTL checking is still in its infancy. Unfortunately, formally proving the equivalence of an RTL against a system-level model is much harder than comparing two RTLs for equivalence, mainly due to the incredible amount of freedom designers have in refining the system-level model to the RTL.

In this paper, we focus on a specific problem related to formal equivalence checking of system-level models against RTL, namely the handling of memories. Black-boxing memories as it is done in RTL-RTL equivalence checking only works if the memory read and write patterns in both designs match exactly. In general, this is not the case in ESL equivalence checking and a more sophisticated memory model that can reason about memory reads and writes is required.

However, similar to register mapping in RTL-RTL verification, the equivalence check can be simplified significantly if the mapping between ESL arrays and RTL memories is considered. Unfortunately, even though most arrays in ESL models are implemented by memories in the RTL, the number of different possible implementations is huge. For efficiency reasons, the memory layout may differ, shadow registers may be used to cache data or a single array may be split into multiple memories in the RTL. In addition to that, timing differences between the system-level model and the RTL usually make it necessary to detect at which points in time the arrays/memories match. Section 3 discusses all the features that a memory model should provide in order to be suitable for system-level equivalence checking.

We rely on the user to provide the information about memory mappings. If the mapping between the memories is known, the basic idea for proving the equivalence of both designs is the following: We assume that both designs start with matching memories/arrays before the computation. Then, both designs perform a single transaction and we check if the results match. Finally, we prove that the memories match again after the transaction, which makes it necessary to reason about the equality of memories. The detailed proof procedure is described in Section 4.

2. RELATED WORK

Theories for reasoning about arrays/memories have been a research subject for a long time [6], [7]. In 2001, Stump et al. [8] presented a paper on an extensional theory of arrays. Extensional theories formalize the intuition that if two arrays store the same value at each array location, they are equivalent. Bradley et al. [1] further extend the theory of arrays by allowing quantifiers. They show that their fragment results in a complete satisfiability proof. Our approach for universal quantifier elimination is based on their work. The application of this memory model in formal equivalence checking has been discussed in many papers, related to pipeline verification [3] and symbolic simulation [9]. Bryant and Velev [2] made use of an efficient memory model in showing that two pipelined microprocessors, when started with matching memories produce the same results.

Some recent papers describe clever encodings of read/write formulas into CNF [4], [5]. For example, Manolios et al. [5] presented an encoding that tries to minimize the number of bits/clauses required for modeling memories. Even though we don't deal with this problem, their ideas can potentially be applied to our approach, too.

3. REQUIREMENTS FOR MEMORIES

In the rest of the paper, we don't distinguish between arrays and memories and refer to both of them as "memories". The relationship between ESL and RTL memories is described by a memory mapping:

Definition: Memory mapping

A memory mapping MM is a universally quantified expression over all memory locations. It relates the contents of one memory to the contents of another memory (or several other memories). The memory mapping is expressed in terms of reads of both memories. *Example:* A one-to-one memory mapping between two memories M_A and M_B , both having 10 elements, is described by the following expression:

$$\text{MM}(M_A, M_B) := \bigvee_{0 \leq i < 10} \text{read}(M_A, i) = \text{read}(M_B, i)$$

In Sections 3.1 and 3.2 we describe memory mappings for the following cases:

- If the layout of both memories differ.
- If a memory in one design is split into multiple memories in the other design.
- If the same data is stored in different addresses in both memories.

Section 3.3 deals with non-mapping related constraints on memories.

3.1 Memory layout differences

Differing memory layouts frequently occur when comparing a design specified in a system-level language (e.g. C++) with a design specified at the register-transfer level. Due to the lack of bit-accurate data-types in the system-level language, a memory element may be represented with more bits than actually needed. Consider for example the two memories M_A and M_B in Fig. 1. M_A is a memory in a C++ design and M_B is a memory in an RTL design. The layout of M_A is determined by the C++ struct 'elem'. Due to the limited number of available data-types in C++, it may be necessary to represent a field in the struct with more bits than actually

necessary. In this particular case, even though only three bits are used per element, they are stored in two character variables, resulting in a total of 16 bit storage. In the RTL memory on the other hand, the corresponding data is stored in a compressed format. The actual mapping between the bits in the C++ memory and the RTL memory is indicated in Fig. 1.

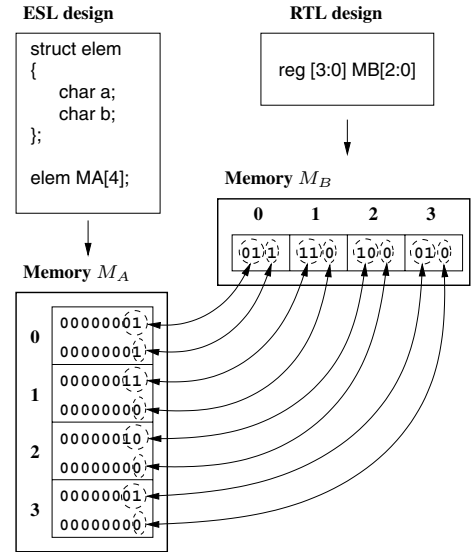


Figure 1: Mapping with layout differences

The mapping can be expressed as

$$\text{MM}(M_A, M_B) := \bigvee_{0 \leq i < 4} \text{read}(M_A, i)[7:0] = \{6'd0, \text{read}(M_B, i)[2:1]\} \wedge \text{read}(M_A, i)[15:8] = \{7'd0, \text{read}(M_B, i)[0]\}$$

Having the user specify such a formula for more complex mappings is very inconvenient. To deal with this issue, we introduce "templates". For memory mappings where the layout within each memory location is different, the user can specify a template which describes the mapping within a single memory location. For the example in Fig. 1, the user would create the following template 't':

```
template t
{
    a = [2:1];
    b = [0];
}
```

With this template, the mapping expression simplifies to:

$$\text{MM}(M_A, M_B) := \bigvee_{0 \leq i < 4} \text{read}(M_A, i) = \text{template t}(\text{read}(M_B, i))$$

3.2 Multiple memories and address mappings

Memory mappings can also be used to relate multiple memories to each other. Also, corresponding data items don't need to be stored at the same address in the memories. Consider for example the mapping in Fig. 2.

The first four memory locations in memories M_E , M_F and M_G are all mapped one-to-one. Location 5 in M_E is mapped to location 4 in M_F and location 6 in M_E is mapped to location 4 in M_G . A case like this sometimes occurs if a single memory in the system-level model (here M_E) is split into multiple memories in the RTL (here

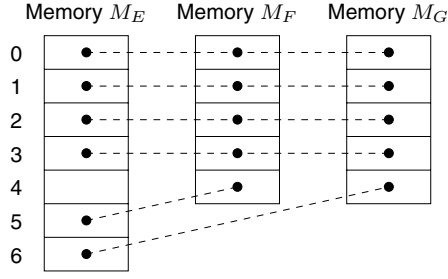


Figure 2: Mapping between multiple memories

M_F and M_G) in order to increase memory access performance. The memory mappings of this example are described by:

$MM(M_F, M_E) :=$

$$\forall_{0 \leq i < 5} \text{read}(M_F, i) = (i < 4) ? \text{read}(M_E, i) : \text{read}(M_E, 5)$$

$MM(M_G, M_E) :=$

$$\forall_{0 \leq i < 5} \text{read}(M_G, i) = (i < 4) ? \text{read}(M_E, i) : \text{read}(M_E, 6)$$

3.3 Constraints on memories

In addition to mappings between memories it can be necessary to allow the user to define constraints on the memory contents, for example if it is known that a memory can only hold certain data-values. Furthermore, a constraint can also become part of a proof obligation, for example, if the correctness of the equivalence check requires that the memory content ends up with a certain set of data-values after the computation. In our approach, the user can either specify constraints on individual memory locations or in a universally quantified form over all memory locations. A constraint may also be expressed over multiple memories.

Examples:

$$c_0 = (\text{read}(M_A, 3) = 2)$$

$$c_1 = \forall_i (\text{read}(M_A, i) < 3)$$

$$c_2 = \forall_i (\text{read}(M_A, i) + \text{read}(M_B, i) > \text{read}(M_C, i))$$

4. PROOF PROCEDURE

A common notion of equivalence between two designs with memories is transaction equivalence: Assuming that the designs start out in a valid state, both designs execute a single transaction. If both of them produce equivalent outputs on the way and end up in a valid state, the designs are considered equivalent.

The proof method we use for proving transaction equivalence is induction. In the induction step, a valid starting state is assumed, the two designs are unrolled for an entire transaction and the outputs at the end of the transaction are compared. Furthermore, the state in which both designs end up is compared against the starting state. The induction base is required if the starting state does not include the initial state. In this case, we additionally need to prove equivalence of the designs until the state becomes a subset of the starting state. In describing the subsequent steps we assume that the initial state is included in the starting state and no induction base is required.

The procedure is depicted in Fig. 3. In this example, a transaction on the ESL model requires two cycles whereas the same transaction requires three cycles on the RTL. The two designs are unrolled

for two (ESL_0, ESL_1) and three time-frames (RTL_0, RTL_1, RTL_2), respectively. Both designs start in a valid state (S_A, M_A) and (S_B, M_B) where S_A, S_B denote the register state and M_A, M_B denote the memory state of the corresponding design. Both designs are fed with equivalent input data. However, due to timing differences between both models, the inputs don't necessarily match one-to-one. For example, the ESL model might receive the entire data at the same time, whereas the RTL model might receive it in serialized form. In Fig. 3 this is indicated by input assignments I which are not connected together. By unrolling both designs, we can construct a formula for the outputs O_A, O_B , as well as for the state (S'_A, M'_A) and (S'_B, M'_B) after the transaction. The designs are considered to be equivalent if $O_A = O_B$ and the combined states are valid states.

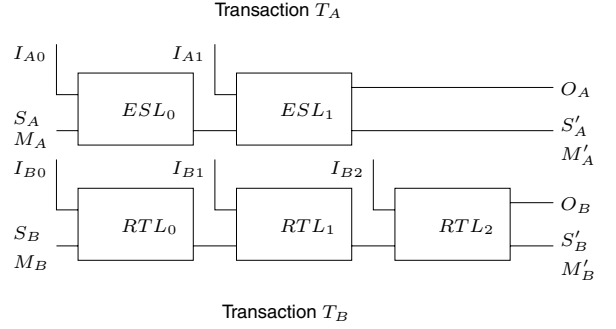


Figure 3: Proof procedure

In order for the induction proof to succeed, additional inductive invariants may be required. For example, even if we assume that the memories start out in the most general state, a successful equivalence check generally requires that their contents match in both designs. Memory mappings and constraints as discussed in Section 3 provide such invariants. From the information the user supplies, i.e., constraints, templates and mappings, we construct a system of assumption formulas:

- Assumptions relating the memories in both designs. These assumptions involve all the user specified memory mappings:
$$a_0 = MM_0(M_A, M_B) \wedge MM_1(M_A, M_B) \wedge \dots$$
- Constraints on the memories:
$$a_1 = c_0(M_A, M_B) \wedge c_1(M_A, M_B) \wedge \dots$$
- Register mappings between the two designs:
$$a_2 = r_0(S_A, S_B) \wedge r_1(S_A, S_B) \wedge \dots$$
- Additional state invariants given by the user involving memories and registers:
$$a_3 = i_0(M_A, M_B, S_A, S_B) \wedge \dots$$

Given these assumptions, the following proof obligations are required for an equivalence check:

- Prove that the memory mappings hold after the transaction:
$$a_0 \wedge a_1 \wedge a_2 \wedge a_3 \Rightarrow MM_0(M'_A, M'_B) \wedge \dots$$
- Prove that the constraints still hold after the transaction:
$$a_0 \wedge a_1 \wedge a_2 \wedge a_3 \Rightarrow c_0(M'_A, M'_B) \wedge c_1(M'_A, M'_B) \wedge \dots$$
- Prove that the register mappings hold after the transaction:
$$a_0 \wedge a_1 \wedge a_2 \wedge a_3 \Rightarrow r_0(S'_A, S'_B) \wedge r_1(S'_A, S'_B) \wedge \dots$$

- Prove that the additional invariants hold after the transaction:
 $a_0 \wedge a_1 \wedge a_2 \wedge a_3 \Rightarrow i_0(M'_A, M'_B, S'_A, S'_B) \wedge \dots$
- Prove that no memory accesses are out-of-bounds. The behavior of an out-of-bounds array access is undefined in most system-level languages like C++. Thus, for a correct equivalence check, we must make sure that this never happens.

Some of these formulas still have universal quantifiers in them which must be removed before they are handed over to a SAT solver. In addition to that, bit-blasting the memories is impractical for most designs. Thus, we use a memory model whose size only depends on the number of accesses, not on the actual size of the memories. The algorithm for removing the quantifiers and converting the memories is outlined here:

1. Propagate reads over writes. This is done by applying the following rewrite rule on the formulas:

$$\text{read}(\text{write}(M, i, d), j) \rightarrow \text{ite}(i = j, d, \text{read}(M, j))$$

where i is the address of the write, d is the written data and j is the address of the read. Applying this rule successively removes all writes to memories in the formulas.

2. Replace universal quantifier variables on the right-hand side of a proof obligation by free variables and drop the quantifier. Because quantifiers can only occur in the top-level conjunction of the right-hand side,

$$a \Rightarrow \forall_i (\text{read}(M_A, i) = \text{read}(M_B, i))$$

is equivalent to

$$a \Rightarrow (\text{read}(M_A, i') = \text{read}(M_B, i'))$$

when given to a validity checker (i' is a fresh free variable). The validity proof only returns true if the formula is satisfied for all assignments to i' . If bounds were given for the quantifier variable, those bounds become part of the assumptions.

3. Expand assumption quantifiers. Removing universal quantifiers on the left-hand side of a proof obligation, i.e., an assumption, can be done by expanding the quantified expression over all possible values of the quantifier variable:

$$\forall_i (\text{read}(M_A, i) = \text{read}(M_B, i)) \Rightarrow p$$

is equivalent to

$$\begin{aligned} \text{read}(M_A, 0) &= \text{read}(M_B, 0) \wedge \\ \text{read}(M_A, 1) &= \text{read}(M_B, 1) \wedge \\ &\vdots \\ \text{read}(M_A, n) &= \text{read}(M_B, n) \Rightarrow p \end{aligned}$$

In practice, it's usually not necessary to expand over all possible values of the quantifier variable. If a specific value is never used in any other read, it can safely be dropped without compromising completeness. In our approach, we use a similar heuristic for computing sufficient quantifier variable values as in [1]. We collect all address expressions that are used in reads on memories, including the fresh variables introduced in removing the right-hand side quantifiers. The assumption quantifiers are then expanded for all those address expressions. Consider for example the following formula:

$$\begin{aligned} &\forall_i (\text{read}(M_A, i) = \text{read}(M_B, i)) \wedge \\ &\forall_j (\text{read}(M_A, j) < 2) \\ &\Rightarrow \text{read}(M_A, 4) + \text{read}(M_B, a) < 3 \end{aligned}$$

Both quantifiers are expanded with the read addresses 'a' and '4':

$$\begin{aligned} \text{read}(M_A, a) &= \text{read}(M_B, a) \wedge \\ \text{read}(M_A, 4) &= \text{read}(M_B, 4) \wedge \\ \text{read}(M_A, a) &< 2 \wedge \\ \text{read}(M_A, 4) &< 2 \\ &\Rightarrow \text{read}(M_A, 4) + \text{read}(M_B, a) < 3 \end{aligned}$$

4. Perform completeness check.

Because we allow general expressions in mappings, the heuristic used in expanding assumption quantifiers may not be complete (but not unsound!). Consider for example the following mapping:

$$\begin{aligned} &\forall_i (\text{read}(M_A, i + 1) = \text{read}(M_B, i + 1)) \wedge \\ &\text{read}(M_B, 1) = 1 \\ &\Rightarrow \text{read}(M_A, 1) = 1 \end{aligned}$$

With the given heuristic, the formula is expanded into

$$\begin{aligned} &(\text{read}(M_A, 2) = \text{read}(M_B, 2)) \wedge \text{read}(M_B, 1) = 1 \\ &\Rightarrow \text{read}(M_A, 1) = 1, \end{aligned}$$

which obviously will produce a false counter-example. The heuristic used in expanding assumption quantifiers is only complete for certain cases. For example, if the address of all reads in quantified expressions is the quantifier variable itself (all reads have the form $\text{read}(M, i)$), the heuristic is complete. In our approach, we detect cases for which we know that the result is complete. For all other cases, we flag a warning to the user. The completeness check is based on the work in [1].

5. Replace reads by free variables. At this point, the formulas are free of quantifiers. The only memory operations still left are reads on memories. In this last step, we replace reads by free variables. Multiplexers are introduced which maintain the equivalence relationships between reads. For example, if there are three reads $\text{read}(M, a)$, $\text{read}(M, b)$, $\text{read}(M, c)$ in the formulas, they are rewritten into:

$$\begin{aligned} \text{read}(M, a) &\rightarrow v_1 \\ \text{read}(M, b) &\rightarrow \text{ite}(b = a, v_1, v_2) \\ \text{read}(M, c) &\rightarrow \text{ite}(c = a, v_1, \text{ite}(c = b, v_2, v_3)) \end{aligned}$$

6. Prove formulas using a validity checker. In our approach, we convert the problem into a satisfiability problem and solve it using a SAT solver.

5. TIMING DIFFERENCES

In the proof procedure in Section 4, we assumed that the memory mappings hold before the transaction is started. This is not always the case. For example, when comparing a non-pipelined ESL design against a pipelined RTL designs, the memories at the beginning of a transaction may not match up because the RTL may still have a previous transaction in-flight which has not yet committed its result to memory. However, both designs can still be equivalent if the memories match up right before the current RTL transaction writes to memory. In essence, this means that there exists a memory mapping between the ESL model and the RTL at different time steps. In the example in Fig. 4, the ESL model computes an entire transaction in a single cycle whereas the RTL is a pipelined design with three stages. The write to memory happens in stage three. If both designs start out with matching memories at time zero, the ESL design already writes its first result at the end

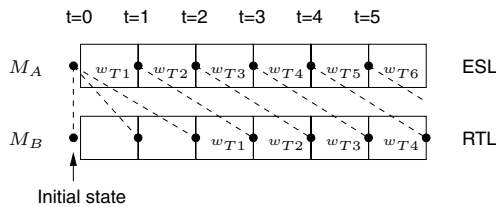


Figure 4: Mapping with timing differences

of the first cycle (w_{T1}) whereas the RTL has only finished the first stage. Thus, memory M_B at time-step 1 is still equivalent to M_A at time-step 0 (depicted by the dashed line in Fig. 4. In the second cycle, the ESL writes yet another result to memory (w_{T2}) whereas in the RTL, the write of the first transaction still hasn't happened yet. Thus, memory M_B at time-step 2 is still equivalent to M_A at time-step 0. Finally, in the third cycle, the RTL writes the first result to memory. From there on, each of the designs produces a new result every cycle, with the difference that the memory M_B at time-step t corresponds to memory M_A at time-step $t-2$.

The proof procedure does not need to be fundamentally changed to handle timing differences. Because we're operating on an unrolled model, we have access to the state of the design in several time frames. The only difference is that our memory mappings now have time as additional parameter.

$$\text{MM}(M_A, M_B, t) := \bigvee_i (\text{read}(M_A(t-2), i) = \text{read}(M_B(t), i))$$

When computing the formulas, it must be confirmed that when referring to M_A and M_B , the correct time frames in the unrolled model are chosen.

Another difficulty that arises is that now the induction base becomes necessary. First of all, we need to ensure that the given mapping really holds after the design has settled and second, we need to make sure that the designs are indeed equivalent in the first few cycles until the mapping stabilizes.

6. EXPERIMENTAL RESULTS

The memory modeling is implemented in our C++ to RTL equivalence checker Hector. It has been used on a number of industrial designs which all contain memories. The C++ as well as the RTL were hand-coded by different designers, so they differ rather substantially. In order to prove the designs equivalent, all of the features of our memory model, i.e., templates, mappings between multiple memories and quantified constraints had to be used. They were provided by the designers who knew exactly how the C++ arrays and the RTL memories were related. Table 1 presents the results. The second and third columns show the number of lines of code of the C++ and the RTL code, respectively. These numbers are approximate numbers because we didn't count code in libraries. The fourth column contains the number of memories. The first number is the number of memories in the C++, the second number is the number of memories in the RTL. Column five shows

Design	#LOC C++	#LOC RTL	#MEMs	#DIS	#BUGS	time	result
D1	50	6200	1/1	0	0	4min	proven
D2	70	580	1/1	0	0	2min	proven
D3	570	1720	3/1	9	2	4min	proven
D4	1700	7500	4/4	8	2	<1h	proven
D5	4300	6700	31/33	>40	4	43min	proven or cex

Table 1: Equivalence check of designs with memories

the number of discrepancies that were found using Hector. Some of the discrepancies were caused by missing constraints, some turned out to be real bugs. The actual number of bugs found is shown in column six. Column seven gives the time required for the proof and column 6 shows the final result. All designs could be proven correct (after fixing the bugs) except design D5. For D5, Hector could prove most of the outputs correct except for a few where it found counter-examples. All output checks were conclusive, though, which means we either got a proof or a counter-example.

7. CONCLUSION

We have discussed several problems that occur when formally checking the equivalence of an RTL model with memories against its ESL specification. Specifically, due to implementation decisions, the memory layouts may differ or constraints may be required which can only be formulated by universally quantified expressions on the memories. We have presented a proof approach for transaction equivalent designs with memories. The idea is to use memory mappings provided by the user as invariants for an induction proof. Finally, we've demonstrated the practicality of the approach by applying it to several industrial designs.

8. REFERENCES

- [1] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *VMCAI*, pages 427–442, 2006.
- [2] R. E. Bryant and M. N. Velev. Verification of pipelined microprocessors by comparing memory execution sequences in symbolic simulation. In *ASIAN '97: Proceedings of the Third Asian Computing Science Conference on Advances in Computing Science*, pages 18–31. Springer-Verlag, 1997.
- [3] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *CAV*, pages 68–80, 1994.
- [4] M. K. Ganai, A. Gupta, and P. Ashar. Verification of embedded memory systems using efficient memory modeling. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1096–1101. IEEE Computer Society, 2005.
- [5] P. Manolios, S. K. Srinivasan, and D. Vroon. Automatic memory reductions for rtl model verification. In *ICCAD 2006: ACM-IEEE International Conference on Computer Aided Design*, 2006.
- [6] J. McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.
- [7] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.
- [8] A. Stump, C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for an extensional theory of arrays. In *Logic in Computer Science*, pages 29–37, 2001.
- [9] M. N. Velev, R. E. Bryant, and A. Jain. Efficient modeling of memory arrays in symbolic simulation. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 388–399. Springer-Verlag, 1997.