Making Complex Timing Relationships Readable: Presburger Formula Simplification Using Don't Cares

http://www.cs.swt.edu/papers/dac98/

Tod Amon[†], Gaetano Borriello[‡], Jiwen Liu[†]

[†]Department of Computer Science
Southwest Texas State University
San Marcos, TX 78666
tod@cs.swt.edu

[‡]Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195
gaetano@cs.washington.edu

Abstract

Solutions to timing relationship analysis problems are often reported using symbolic variables and inequalities which specify linear relationships between the variables. Complex relationships can be expressed using Presburger formulas which allow Boolean relations to be specified between the inequalities. This paper develops and applies a highly effective simplification approach for Presburger formulas based on logic minimization techniques.

1 Introduction

Many problems in computer-aided design (ranging from physical placement to scheduling in behavioral synthesis) can be formulated using systems of linear inequalities. This is especially the case for problems that deal with temporal information. In the simplest cases, the inequalities must all hold and are derived from the delay ranges in the specification and the constraints (imposed by the user or environment) that must be satisfied for the system to function correctly. Graph algorithms have been effectively applied to determine the satisfiability (the ranges of allowable values for each variable) of these systems of inequalities [7, 13].

In more complex cases, the temporal relationships may involve minimums and maximums which often lead to disjoint solution spaces and disjunctions between the linear constraints (e.g., `{[X,Y]: X <= 30 or Y <= 30 or X + Y <= 70}`)[1, 14]. Presburger formulas – which consist of affine constraints over integer variables, the logical connectives $\neg$, $\wedge$, $\vee$, and the quantifiers $\forall$ and $\exists$, can be used to specify these types of relationships.

Although Presburger formulas have not been widely used due to complexity concerns, they have been effectively used for problems such as dependence analysis for advanced compiler optimizations [9], model checking of infinite state programs [3], and mechanical verification in theorem provers [4]. Some of the more recent uses have arisen because new software that can efficiently manipulate Presburger formulas has been developed (i.e., the Omega libraries [5, 6, 10, 11]).

We have recently been using this software to perform symbolic timing verification of timing diagrams [2]. The promise of symbolic timing verification is that it can be used to derive constraints to be imposed on synthesis tools and/or enable the abstraction of complex timing relationships. Working with symbolic variables can also help humans understand and validate formal specifications of timing behavior.

One problem is that the formulas reported by the Omega tools (and we suspect other libraries which support Presburger formulas) are in some cases not easy to read. This is a problem both for a human designer that may be trying to interpret the result and make design decisions as well as for synthesis tools that may work inefficiently with a large Presburger formula that relates many variables (i.e., delays in the circuit) with each other. In many cases, the results reported by Omega can be simplified. Omega makes use of some important simplification strategies but these are often ineffective even on formulas that a human can simplify in a few minutes. Many formulas require several hours of human manipulation before reaching their simplest form.

In this paper, we report on a technique we have developed to automatically simplify quantifier free Presburger formulas obtained using the Omega libraries (our symbolic timing verification work results in these types of formulas) using techniques from logic minimization that exploit don't care information. Our results to date have been quite promising and we believe argue strongly that future implementations of Omega should support this simplification methodology. Because the formulas can be simplified efficiently, our results further support our belief that Presburger formulas can be effectively used to express and verify timing constraints for a variety of applications.

Section 2 briefly describes how we use Presburger formulas to perform symbolic timing verification and presents examples which help motivate our work. Section 3 describes our methodology for simplifying Presburger formulas and how we derive the crucial don't care information needed for minimization. Section 4 presents our results including examples of the simplifications our tools can accomplish and the execution times required. Section 5 presents our conclusions.

2 Symbolic Verification of Timing Diagrams

We have been using Presburger formulas to perform symbolic verification of timing diagrams, e.g., see Figure 1. Because we include support for symbolic variables, our verification problem consists of determining the requirements that the symbolic variables must meet in order for the timing constraints to be satisfied assuming that the other temporal relationships (e.g., specified propagation delays) are maintained.

The problem description takes the form of a set of integer n-tuples where n is the number of symbolic variables. Within the formula we quantify variables which represent times at which signal transitions take place and use inequalities to express all of the temporal relationships (see [2] for details). We use the Omega libraries to remove the quantifications (using Fourier-Motzkin variable elim-

DAC 98, San Francisco, California

ination) to obtain a description of the requirements that the symbolic variables must meet in order for the constraints to always be satisfied. Some of these requirements are quite trivial while others represent important discoveries that arise from transforming the specification by removing the quantifications using Omega. Numeric bounds for the symbolic variables may result, or important relationships between the symbolic variables may be exposed. We present two examples.

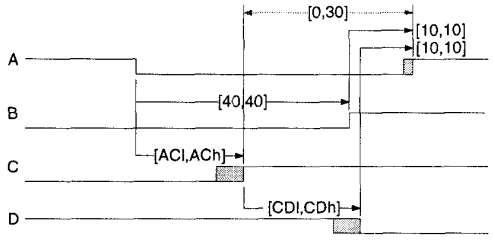


Figure 1

A timing diagram with a single constraint requiring that the second edge on signal A occurs no more than 30 nanoseconds after the first edge on signal C. All other times (in the range of [low,high]) are propagation delays some of which are "unknowns" expressed using symbolic variables. Presburger formulas are used to formally capture the semantics of the timing diagram. The Omega libraries are used to help perform verification.

2.1 A Simple Example

Figure 1 contains a simple example taken from [1] in which CLP(R) and symbolic linear programming are used to perform the verification. Using our methodology the timing diagram is analyzed and the following integer tuple set is obtained:

```
{ [ACl,ACh,CDl,CDh]:
Exists (A_1,A_2,B_1,C_1,D_1:
 0 <= ACl <= ACh and 0 <= CDl <= CDh and A_2 >= A_1 and
    ACl <= (C_1-A_1) <= ACh and
     40 <= (B_1-A_1) <= 40 and
    CDl <= (D_1-C_1) <= CDh and
  max(B_1+10,D_1+10) <= A_2 <= max(B_1+10,D_1+10)) and
Forall (A_1,A_2,B_1,C_1,D_1: not (
 0 <= ACl <= ACh and 0 <= CDl <= CDh and A_2 >= A_1 and
    ACl <= (C_1-A_1) <= ACh and
     40 <= (B_1-A_1) <= 40 and
    CDl <= (D_1-C_1) <= CDh and
 max(B_1+10,D_1+10) <= A_2 <= max(B_1+10,D_1+10)
) or (0 <= (A_2-C_1) <= 30  ))
};
```

Our verification tools transform the above formula into a Presburger formula (by removing the maximizations) and then use the Omega libraries to remove all of the quantified variables (i.e., the events in the timing diagram). The result is a greatly simplified Presburger formula which is obtained using .2 seconds of CPU time and specifies the requirements that the symbolic variables must meet in order to satisfy the constraint (i.e., $0 \leq A_2 - C_1 \leq 30$):

```
{[ACl,ACh,CDl,CDh]: 20 <= ACl <= ACh <= -CDh+39 and
                     0 <= CDl <= CDh) union
{[ACl,ACh,CDl,CDh]:  0 <= CDl <= CDh <= 20 and
                    20 <= ACl <= ACh and
                    40 <= ACh+CDh}
```

We can, however, use Omega to formally prove that this solution is equivalent to a solution obtained by manual inspection:

```
{[ACl,ACh,CDl,CDh]:  0 <= CDl <= CDh <= 20 and
                    20 <= ACl <= ACh}
```

Interestingly enough, we have on occasion found that the Omega libraries can produce this simplified answer. There are some simplification algorithms built into Omega but they are not as general as the one we describe in this paper. Omega always presents solutions as unions of sets containing no disjuncts (i.e., in sum of products form).

2.2 Interface Verification Example

This second example is taken from [15] which contains two specifications for a memory controller and a bus. Here, we consider only the read protocol and the combined specification as shown in Figure 2. Using symbolic variables we can easily perform experiments which might, for example, help us to select another memory controller. We have added a constraint (i.e., ResponseTime) to the specification to ask the question: "Given these values (some of which are symbolic) what will the overall length of the read cycle be?"

Omega reports (using .6 seconds of CPU time to analyze a fairly complex Presburger formula) an answer that is difficult to read. One extension we have added to our tools is the ability to *map* Presburger formulas by introducing *mapped variables* in place of inequalities. Our tool can also replace unions by disjunctions. Using these simple tools, we can report the result in a more readable form as:

```
{[ RRl , RRh , PWl , PWh , RTl , RTh ]: (
( a and  b and c and d and  e and f and g and h ) or
( i and  b and c and h and !f and d and g ) or
( b and !a and c and e and  f and g ) or
( b and  c and j and d and  k and e and  f and g ) or
( i and  b and c and j and  d and e and !f and g ))}
map
a : RTl <= RRl + 160
b : RRl >= 30
c : RRh >= RRl
d : PWl >= 0
e : PWh >= PWl
f : RTl <= 10 + RRl + PWl
g : RTh >= 30 + RRh + PWh
h : PWh >= 150
i : RTl <= RRl + 150
j : RTh >= 180 + RRh
k : PWl <= 150
```

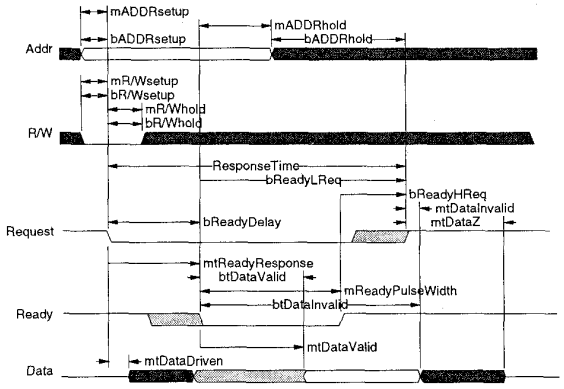
By inspection one can note that many of the inequalities appear multiple times in the result. One of our original intentions was to use a logic minimization tool to support factoring and algebraic manipulation of the resulting formulas. This can help present answers in a more readable form and also help one to manually simplify the formula. Over the course of several days spent working with this resulting formula, we obtained a much simpler formula which Omega verified was identical to the result reported above (the same mapping applies):

```
{[ RRl , RRh , PWl , PWh , RTl , RTh ]:
b and c and d and e and g and j and (f  or ( i and k)) }
```

Automatically obtaining simplified answers such as this one is the purpose of the work reported in this paper. Using simple answers one can more easily and efficiently uncover important and interesting relationships than would be possible with the more complex results reported by Omega.

3 Simplification Methodology

Our simplification methodology is based upon mapping Presburger formulas and then extracting don't care information so that we can better use logic minimization tools such as SIS (Espresso) to perform the actual simplification. Using this technique we can obtain



Type	Name	Bounds
constraint	mADDRsetup	[25,]
constraint	mR/Wsetup	[25,]
constraint	mADDRhold	[15,]
delay	mtReadyResponse	[RRl,RRh]
delay	mtDataValid	[40,90]
guarantee	mtDataDriven	[5,]
guarantee	mReadyPulseWidth	[PWl, PWh]
guarantee	mtDataInvalid	[0,]
guarantee	mtDataZ	[0,5]
guarantee	bADDRsetup	[30,]
guarantee	bR/Wsetup	[30,]
guarantee	bADDRhold	[20,]
guarantee	bR/Whold	[20,]
constraint	btDataValid	[,100]
constraint	btDataInvalid	[140,]
constraint	bReadyDelay	[30,]
delay	bReadyLReq	[150,180]
delay	bReadyHReq	[10,30]
constraint	ResponseTime	[RTl,RTh]

Figure 2

Combined timing diagram and tables for the bus and memory controller. The prefixes "b" and "m" indicate whether the relationship came from the bus specification or the memory specification. The response time constraint was added after the diagrams were composed.

not only mimized sum-of-products form but also factored forms that tend to be more readable because the number of literals is minimized.

As a simple example, consider the Presburger formula: { [X] : X > 20 and X > 30} which should obviously be simplified and reported as { [X] : X > 30}. Omega contains algorithms for eliminating redundant constraints and would in fact report this solution. We map this formula to produce:

```
{ [X]: (a and b) }
map
a : X > 20
b : X > 30
```

and then discover the don't care: $\mathbf{a}'\,\mathbf{b}$ (i.e., **not a and b**) which when included for the minimization of $\mathbf{a}\,\mathbf{b}$ (i.e., $\mathbf{a}\,\mathbf{b} + \mathbf{a}'\,\mathbf{b}$) produces **b** which corresponds to { [X] : X > 30}.

The don't cares we discover are reported as a mapped Presburger formula in sum-of-products form (each product is a don't care). Discovering this information is potentially quite time consuming. Our initial efforts have been based upon the rather obvious and rather inefficient method of considering every possible *combination* of mapped variables. If there are M mapped variables and one wishes to consider don't cares containing n mapped variables there are: $\binom{M}{n}$ combinations. Each combination represents 2^n potential don't cares because each mapped variable might appear in a don't care in either negated or non-negated form. Each potential don't care is analyzed using Omega which will report *false* if there is no way to satisfy the conjuncted inequalities (e.g., { [X] : not (X > 20) and X > 30 }.

We have identified two optimizations we believe are quite important and useful for making this computationally feasible. We also believe other optimizations and improvements to our algorithm are quite likely to be discovered (we intend to investigate a more constructive approach to finding don't cares which would avoid the need to even generate all of the combinations).

One optimization uses previously discovered don't cares to avoid generating don't cares which are of no interest (e.g., if $\mathbf{a}\,\mathbf{b}'$ is a don't care then obviously so is $\mathbf{a}\,\mathbf{b}'\,\mathbf{x}$ where $\mathbf{x}$ is any mapped variable). For the examples we looked at, this optimization does not greatly reduce the time required to extract don't cares because it requires a fair amount of computation time to implement. It does, however, reduce the amount of time needed during logic minimization because fewer don't cares are identified.

Our most important optimization relies on an array that indicates which symbolic variables appear in each mapped variable. If there are M mapped variables and S symbolic variables we construct IN, an $M \times S$ array where:

$$IN[m][s] = \begin{cases} 1 & \text{if mapped variable } m \text{ includes} \\ & \text{symbolic variable } s \text{ in its inequality} \\ 0 & \text{otherwise} \end{cases}$$

This array is consulted for each combination of mapped variables to determine whether or not we need to use this combination to generate potential don't cares. Although we do not implement our algorithm this way, we essentially compute the sum of each row corresponding to a mapped variable in the combination being considered. The resulting vector indicates how many times each symbolic variable appears in the combination.

We do not construct potential don't cares for combinations that have a 1 somewhere in their summation vector. This optimization is based on the observation that if a symbolic variable appears only once then there is only one inequality constraining that variable and thus it is essentially a free variable. Given the presence of a free variable, Omega will not report *false*, unless irrespective of the value of the variable, the result is false (and thus at least one mapped variable is not needed in order to cover this don't care).

Our simplification methodology allows users to specify a range of *don't care levels*, to specify the minimum and maximum number of mapped variables appearing in potential don't cares. Users can adopt a simplification methodology based upon iteratively increasing the maximum don't care level until a reasonable solution is found or compute times grow to be too large.

We have two tools we use to move between mapped Presburger formulas and SIS, a logic optimization tool developed at UC Berkeley [12]. Our tool *eqnextract* takes a mapped Presburger formula as input and produces a `.eqn` file for use by SIS. Our tool *eqninsert* takes a `.eqn` file and a mapped Presburger formula and replaces the formula with that specified by the `.eqn` file.

Using SIS we can read an `.eqn` file and write the result back in the `.blif` format. We have a tool, *blifmerge*, which merges two `.blif` files assuming the second file specifies the onset of the don't cares for the first (i.e., we use the `.exdc` directive). We can then use SIS (Espresso) to read the combined blif file and minimize the function using the don't care information (i.e., using `full_simplify`). If a factored form is desired it can be created and the minimized logic can be written back to an `.eqn` file and then incorporated back into a Presburger formula. Although the underlying minimization metric used by SIS is geared towards reducing the size of the resulting logic we believe it performs a reasonable job of simplifying expressions for human readability.

4 Results

In this section we describe the result of applying our techniques to the examples described in Section 2. Our execution times are quite reasonable especially given that the code we have written is not very efficient and that we are running on a fairly old DEC 3000 workstation with 32 MB of memory. Even more importantly, our algorithms sit on top of Omega. We maintain an internal data structure similar to that used by Omega and a great deal of time is spenting copying between our data structure and the Omega data structure. If our methodology were to be implemented within the Omega libraries vast improvements in execution time should result. Mapping and logic minimization are quite efficient and took less than one second of CPU time for all of the examples reported below.

4.1 Interface Timing Verification

For the mapped formula in Section 2.2 we have:

$$IN = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

The combination **a f k** would result in the summation of rows 1, 6, and 11 and produce:

$$\begin{pmatrix} 2 & 0 & 2 & 0 & 2 & 0 \end{pmatrix}$$

which indicates that symbolic variables `RRl`, `PWl`, and `RTl` each appear twice in the mapped inequalities of **a f k** and that the symbolic variables `RRh`, `PWh`, and `RTh` do not appear at all. Because no symbolic variable appears only once the combination **a f k** would be used to generate potential don't cares. Figure 3 summarizes the results of applying our don't care extraction methodology at various don't care levels. Using the results from don't care level 2, SIS simplified the formula but it still consisted of 5 product terms. Much to our surprise, using the additional results from don't care level 3, SIS reports an even better result than the one we obtained by hand:

```
{[ RRl , RRh , PWl , PWh , RTl , RTh ]:
b and c and d and e and g and j and (f  or i) }
```

4.2 Our Simple Example with More Symbolic Variables

As a test of our methodology we complicated our first example (see Figure 1) by introducing more symbolic variables. Specifically, all numeric values were replaced with symbolic variables including the upper bound (i.e., 30) of our constraint (we kept the lower bound of 0 because we are assuming $A_2 \geq C_1$). Using the Omega libraries to analyze this revised problem (which now contains 11 symbolic variables) we obtained the following solution (after mapping):

```
{[ ACl, ACh, ABl, ABh, CDl, CDh, DAl, DAh, BAl, BAh, CAh ]:
((a and b and c and d and e and f and g
    and h and i and j and k and l) or
 (a and b and c and d and e and f and g
    and h and i and j and m and n) or
 (a and b and c and d and e and f and g
    and h and i and j and !m and l and !k and n))}
map
```

level	analyzed	Omega	time	don't cares
2	2 / 55	8 / 220	0.04	$\mathbf{a' i + d' k'}$
3	8 / 165	62 / 1,320	0.2	$\mathbf{a f' k' + a' f k + d' f i' + e h' k' + e' h k + g h j' + g' h' j}$
4	14 / 330	174 / 5,280	.72	none
5	23 / 462	495 / 14,784	5.12	none
6	35 / 462	1496 / 29,568	17.24	none

Figure 3

Results for the example in Section 2.2 reported for each don't care level. We report the actual number of combinations that needed to be analyzed (based on our optimization) compared to the total number of combinations. We also report the number of Omega calls that were made compared to the number that would have been made if all of the combinations were considered. The CPU time and the don't cares discovered at each level are also reported.

```
a : ACl >= 0
b : ACh >= ACl
c : ABl >= 0
d : ABh >= ABl
e : CDl >= 0
f : CDh >= CDl
g : DAl >= 0
h : DAh >= DAl
i : BAl >= 0
j : BAh >= BAl
k : ACh + CDh < ABl + BAh - DAh
l : ACl >= ABh + BAh - CAh
m : ACl + CDl > ABh + BAh - DAh
n : CDh <= CAh - DAh
```

Don't care analysis at levels 2 and 3 revealed nothing. Don't care analysis at level 4 yielded two don't cares and SIS reduced the answer to two product terms. Don't care analysis at level 5 yielded twelve additional don't cares and using SIS we obtained the following simplified answer:

```
{[ ACl, ACh, ABl, ABh, CDl, CDh, DAl, DAh, BAl, BAh, CAh ]:
(a and b and c and d and e and f and g and
 h and i and j and l and n)}
```

Note that some of the mapped inequalities (i.e., **k** and **m**) disappear as a result of the simplification. The result is much easier to understand, because most of the inequalities are quite trivial (i.e., **a** to **j** simply state that the lower bounds on the delay ranges are greater than zero and that the upper bounds are greater than the lower bounds). The two non-trivial inequalities reflect the essence of the solution: in order for our constraint from C_1 to A_2 to be met we need an upper bound on the delay from C_1 to D_1 and D_1 to A_2 (i.e., inequality **n**) and we need a lower bound on the delay from A_1 to C_1 (i.e., inequality **l**) to ensure that C_1 does not occur too soon with respect to B_1. Readers may want to compare this solution to that reported in Section 2.1 (i.e., where `CAh=30`, `ABh=40`, `DAh=10`, and `BAh=10`) and note that this symbolic analysis tells us which numeric values were important in our original solution.

Our optimization worked particularly well for this example. At don't care level 4 we had to analyze one combination out of a possible 1,001 and at don't care level 5 we had to analyze four combinations out of a possible 2,002. The required execution times for this analysis were thus quite minimal. We believe this example demonstrates the benefits of using symbolic variables to obtain symbolic solutions to timing verification problems. Symbolic solutions can assist designers in evaluating tradeoffs and make it easier to validate and understand their formal specifications.

5 Conclusions

The best known upper bound on the performance of an algorithm for verifying Presburger formulas is $O(2^{2^{2^n}})$ [8] and Omega may have an even larger upper bound. For this reason, Presburger formulas have not been used very much for timing analysis. We believe that this will change because the Omega libraries have demonstrated that manipulations can often be efficiently performed and because the type of analysis that can be provided is quite useful. Our simplification algorithms also have terrible worst case complexity yet are quite capable (given simple optimizations) of greatly improving the results reported by Omega using reasonable amounts of computation.

We intend to make immediate use of our results to help us continue exploring symbolic timing verification of timing diagrams and other timing abstractions. Our simplification methodology will quite likely be of great benefit in other application areas as well, both for users of Omega (i.e., as of February 1994 over 400 researchers had obtained copies of the Omega software libraries [10]) and for other users of Presburger formulas (many theorem provers contain algorithms for analyzing Presburger formulas).

Our tools need to be further refined in addition to being made more efficient. The refinements are needed not only to more automatically translate between forms but also to give users a suite of manipulation routines so that results can be presented in the form most appropriate for the domain of application and intended use. For example, in interface circuits, it might be especially valuable to identify "trivial" inequalities (such as those that require lower bounds to be smaller than upper bounds or that require delays to be greater than zero) and provide users the option of omitting them from displayed solutions. Efficiency improvements revolve around identifying further optimizations in don't care extraction and more tight integration with the Omega libraries.

6 Acknowledgements

This work was supported by an NSF RIA Award (MIP-9410279). We wish to thank Wilbon Davis for several interesting discussions and Ellen Sentovich for help with SIS.

References

[1] T. Amon and G. Borriello. An approach to symbolic timing verification. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, June 1992.

[2] T. Amon, G. Borriello, D. Hu, and J. Liu. Symbolic timing verification of timing diagrams using presburger formulas. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, June 1997.

[3] Tevfik Bultan, Richard Gerber, and William Pugh. Symbolic model checking of infinite state programs using presburger arithmetic. Technical Report UMIACS-TR-96-66, University of Maryland, September 1996.

[4] D. C. Cooper. Programs for mechanical program verification. In *Machine Intelligence 6*, 1971.

[5] Wayne Kelly and William Pugh. Determining schedules based on performance estimation. In *Parallel Processing Letters*, September 1994.

[6] Wayne Kelly and William Pugh. Finding legal reordering transformations using mappings. In *Seventh International Workshop on Languages and Compilers for Parallel Computing*, August 1994.

[7] K. L. McMillan and D. L. Dill. Algorithms for interface timing verification. In *Proc. International Conf. Computer Design (ICCD)*, October 1992.

[8] Derek Oppen. A $2^{2^{2^{pn}}}$ upper bound on the complexity of presburger arithmetic. In *Journal of Computer and System Sciences*, July 1978.

[9] William Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8), August 1992.

[10] William Pugh et al. The omega project. In *URL: http://www.cs.umd.edu/projects/omega*.

[11] William Pugh and David Wonnacott. An exact method for the analysis of value-based array data dependences. In *Proc. 6th Workshop on Programming Languages and Compilers for Parallel Computing*, August 1993.

[12] E. M. Sentovich et al. SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, UC Berkeley, May 1992. see http://www-cad.eecs.berkeley.edu/Software/.

[13] P. Vanbekbergen, G. Goossens, and H. De Man. Specification and analysis of timing constraints in signal transition graphs. In *European Design Automation Conference*, March 1992.

[14] E. Walkup and G. Borriello. Interface timing verification with application to synthesis. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, June 1994.

[15] Elizabeth A. Walkup. *Optimization of Linear Max-Plus Systems with Application to Timing Analysis*. Ph.D. thesis, University of Washington, 1995.