Symbolic Timing Verification of Timing Diagrams using Presburger Formulas

Tod Amon[†], Gaetano Borriello[‡], Taokuan Hu[†], 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

We present a novel set of tools for performing symbolic timing verification of timing diagrams. The tools are multi-purpose with uses in verification, derivation of synthesis constraints, and design evaluation. Our methodology is based on using techniques for manipulating Presburger formulas. We demonstrate using several interesting examples that the method is efficient in practice and should be considered for inclusion in commercial tools.

1 Introduction

Two important areas of design automation are *synthesis* – the process of transforming abstract specifications into physical implementation, and *verification* – the process of formally ensuring designs have been properly implemented and will meet their requirements.

In this paper we present novel work in the area of symbolic timing verification[1] for timing diagrams. Timing diagrams, e.g., see Figure 1, are quite popular (especially for interfaces) and are easy to understand because they correspond to execution snapshots where the time axis is explicit. Symbolic timing verification is a powerful extension to traditional constraint checking in that it allows propagation delays and timing constraints to be specified using variables as well as values. Because of the presence of these variables which are essentially "unknowns," verification will most likely be successful only if these variables are constrained appropriately. Symbolic timing verification has three principle uses:

- implementation verification – which confirms that an implementation of a design and its associated delays will meet the constraints in the original specification,
- derivation of constraints for synthesis – using symbolic delay variables, it is possible to determine the degree of flexibility that is available while still satisfying the timing constraints (this can lead to a more efficient use of resources in the final implementation). Also,
- design evaluation – performed by using symbolic constraint variables and determining bounds on the values of these variables given circuit delays (this can provide information about how well a design will perform and also relate the constraint variables to circuit delays).

[0]This work was supported by an NSF RIA Award (MIP-9410279).

[1]Sometimes the term "symbolic verification" is used to describe a verification methodology (e.g., using BDDs to symbolically explore a state space) but, in our case, the actual input and output make use of symbolic variables.

We believe that symbolic timing verification is a valuable paradigm for thinking about synthesis process because it is a formal approach to some of the problems in design space exploration. Working with symbolic variables instead of numeric values clearly complicates the process of verification, and thus symbolic verification is, generally speaking, a more difficult problem than either synthesis or verification alone.

In this paper, we present a complete suite of verification tools which support symbolic timing verification for a large class of timing diagrams. Section 2 describes timing diagrams and their formal representation as inequalities. Section 3 describes our verification methodology. Section 4 presents several examples and discusses related work. Section 5 examines complexity issues.

2 Timing Diagrams

Timing diagrams are often regarded as being too informal to be used as a formal specification for verification purposes. The existence of formalized timing diagram editors has helped to improve this situation by enforcing a convention on timing diagram notation along with well-defined semantics for constraints.

Although timing diagrams are graphical representations, they inherently describe relationships between pairs of *events* in the timing diagram. An event is represented as an edge in the diagram – a transition from one state (or logic level) to another for a specific signal (waveform). The relationship between two events consists of specifying minimum (δ) and/or maximum (Δ) times of separation.

One of our verification tools, *tdformulize*, is responsible for taking the information present in a timing diagram and converting it into a representation containing inequalities which formally describe the semantics of the diagram. We are currently using one of several commercially available timing diagram editors, "TimingDesigner," from Chronology Corporation [7]. We could easily use another editor for input as long as it captures the appropriate semantic information present in a timing diagram. We refer readers to [3, 18, 19] for more general discussions regarding the formal semantics of timing diagram specifications and to [2, 4, 10, 12] for some specific examples.

Timing Designer allows users to specify three types of timing relationships: *delays*, *guarantees*, and *constraints*. Delays represent a causal relationship between two edges, and guarantees specify relationships which are guaranteed to be maintained (an environment might specify a separation between two input events that it guarantees will not be violated). Constraints are requirements which the circuit should meet (i.e., what its environment expects). For example, designers often specify setup and hold constraints and want to know if they will be met. Every guarantee or constraint from event e_1 to event e_2 with minimum separation δ and maximum separation Δ (henceforth we write $[\delta, \Delta]$) corresponds to the inequality:

$$e_1 + \delta \leq e_2 \leq e_1 + \Delta.$$

The formal semantics for delays are a bit different, because they represent causal information. If an event (edge) has two or more incident delay relationships, a maximum is introduced into the equation (essentially stating that the event will not happen until both of the events it is dependent upon have occurred). Thus, delays e_1 to e_3 with separations $[\delta, \Delta]$ and e_2 to e_3 with separations $[\delta', \Delta']$ results in the inequality:

$$max(e_1 + \delta, e_2 + \delta') \leq e_3 \leq max(e_1 + \Delta, e_2 + \Delta').$$

The maximization would include more than two terms if more than two delays were incident on e_3. This is the essential difference between delays and guarantees. Delays, because they are of a causal nature, may result in maximum constraints, whereas guarantees do not. Our verification methodology can easily support minimum constraints as well, but we omit them from consideration in this paper (most existing work in the area of interface verification has focused on maximum constraints which are often sufficient). Note that we do not require both δ and Δ to be present, and thus sometimes the inequalities are one-sided.

Symbolic variables may be used in place of numeric values for δ or Δ for any of the three types of timing relationships. Because Timing Designer does not directly support symbolic variables, we make use of the ability to name each timing relationship. Specifically, any name for a guarantee, delay, or constraint which ends with the special text "%X%Y" is assumed, for the purpose of the inequalities, to have $\delta = X$ and $\Delta = Y$ (either X or Y can be omitted). This way of adding symbolic variables to Timing Designer is advantageous in that it solves the problem of deciding how to draw the timing diagram when some of the $[\delta, \Delta]$ values are symbolic. In this case, the numeric values of δ and Δ can be viewed as hints (or guesses) which Timing Designer can use to determine how to draw the timing diagram. Eventually, the results of verification could be fed back into the timing diagram and affect these numeric values and, thus, the position of signal transitions.

3 Verification

The first verification step consists of using the tool *tdformulize* to produce a formal description of the requirements that the symbolic variables must meet in order for the constraints to be satisfied whenever the delays and guarantees are met. The description takes the form of a set of integer n-tuples where n is the number of symbolic variables:

$$\{[symbolic\ variables] : tdformula\}.$$

All of the delays, constraints, and guarantees present in the timing diagram are present in *tdformula*. Additional inequalities implied by the semantics of the timing diagram are also added by *tdformulize*. For example, we add default guarantees which: order all of the events taking place on the same signal (i.e., $e_1 \leq e_2$), specify that $\delta \leq \Delta$, and for delay relationships that $\delta \geq 0$. The *tdformula* contains two quantifications of event variables which represent integer values for times at which the events in the timing diagram might take place:

$$(\ \forall events :\ \ constraints\ \vee\ \neg(delays\ \wedge\ guarantees)\ )\ \wedge$$
$$(\ \exists events : delays \wedge guarantees\ ).$$

The first line of the formula contains a universal quantification which states that for all possible assignments of times to the events, either the constraints must be satisfied or the assignment must not be well formed (because $delays \wedge guarantees = false$). The second line of the formula ensures that the symbolic variables must take on values which are indicative of a well formed problem.

The next step in the verification process is performed by our tool *tdverify* which performs a number of small simple transformations on *tdformula* in order to obtain a nicely structured Presburger formula. Presburger formulas consist of affine constraints over integer variables, the logical connectives $\neg$, $\wedge$, $\vee$, and the quantifiers $\forall$ and $\exists$. One of the important transformations performed by *tdverify* results in the removal of the maximizations from the inequalities using:

$$max(\alpha, \beta) \leq \gamma \equiv \alpha \leq \gamma \wedge \beta \leq \gamma$$
$$max(\alpha, \beta) \geq \gamma \equiv (\alpha \geq \beta \wedge \alpha \geq \gamma) \vee (\beta \geq \alpha \wedge \beta \geq \gamma).$$

After performing these transformations, *tdverify* makes extensive use of the Omega libraries [16], a set of software recently developed at the University of Maryland for manipulating (among other things) integer tuple sets described using Presburger formulas. The Omega libraries are used to remove the quantifications from *tdformula* and produce as a result:

$$\{[symbolic\ variables] : answer\}.$$

The answer may be *false* – indicating that there is no way to ensure that the *constraints* will be satisfied. Otherwise, the answer consists of a description of the requirements that the symbolic variables must meet in order for the constraints to always be satisfied whenever the delays and guarantees are met. Some of these requirements are quite trivial (e.g., $\delta \leq \Delta$) 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. This is the essence of our work; discovering relationships between the circuit's delays, guarantees, and constraints.

4 Examples and Related Work

In this section, we present several verification examples and discuss the most relevant related work. For all examples we report execution times for *tdverify* in CPU seconds on a DEC 3000 with 64 MB of memory.

4.1 A simple symbolic example

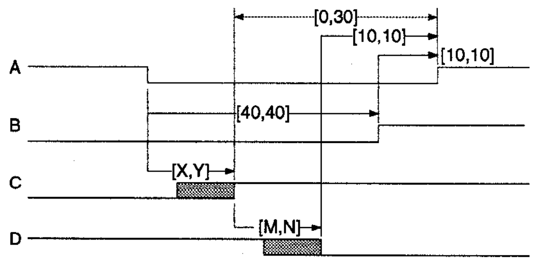


Figure 1

A timing diagram with a single constraint requiring that the second edge on signal A (event A_2) be within $[0, 30]$ of the first edge on signal C (event C_1). All other times are propagation delays.

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 (which contains five events: A_1, A_2, B_1, C_1, D_1) is analyzed and the following integer tuple set is obtained by *tdformulize*:

$$
\begin{array}{rcccl}
\multicolumn{5}{l}{\{[M,N,X,Y]:} \\
\multicolumn{5}{l}{\quad(\forall A_1,A_2,B_1,C_1,D_1:} \\
C_1 & \leq & A_2 & \leq & C_1+30 \quad \vee \\
\multicolumn{5}{l}{\quad \neg(A_1 \leq A_2 \ \wedge\ 0 \leq X \leq Y \ \wedge\ 0 \leq M \leq N \ \wedge} \\
max(B_1+10, D_1+10) & \leq & A_2 & \leq & max(B_1+10, D_1+10) \quad \wedge \\
A_1+40 & \leq & B_1 & \leq & A_1+40 \quad \wedge \\
A_1+X & \leq & C_1 & \leq & A_1+Y \quad \wedge \\
C_1+M & \leq & D_1 & \leq & C_1+N \quad)\,) \ \wedge \\
\multicolumn{5}{l}{\quad(\exists A_1,A_2,B_1,C_1,D_1:} \\
\multicolumn{5}{l}{\quad A_1 \leq A_2 \ \wedge\ 0 \leq X \leq Y \ \wedge\ 0 \leq M \leq N \ \wedge} \\
max(B_1+10, D_1+10) & \leq & A_2 & \leq & max(B_1+10, D_1+10) \quad \wedge \\
A_1+40 & \leq & B_1 & \leq & A_1+40 \quad \wedge \\
A_1+X & \leq & C_1 & \leq & A_1+Y \quad \wedge \\
C_1+M & \leq & D_1 & \leq & C_1+N \quad)\,\}.
\end{array}
$$

Our verification tool *tdverify* transforms the above formula into a Presburger formula (by removing the maximizations) and then uses the Omega libraries to simplify the Presburger formula and remove all of the quantified variables (the events). 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 ($C_1 \leq A_2 \leq C_1 + 30$) assuming the delays and guarantees are met:

$$\{[M,N,X,Y]:\ 0 \leq M \leq N \leq 20 \ \wedge\ 20 \leq X \leq Y\}.$$

The solution can be explained by observing that the constraint would obviously be violated if $N > 20$ and because A_2 waits for both B_1 and D_1 the constraint would also be violated if $X < 20$.

4.2 Event graphs

There is a very direct relationship between our verification work and previous work regarding the verification of *event graphs*. An event graph is an alternative way of representing a set of inequalities, and thus our tools support the verification of many different types of event graphs. Typically constraints are not present in an event graph. The event graph is used to determine minimum and maximum separations between pairs of events and these calculations are then compared with any required constraints. If no guarantees are present, a simple linear time algorithm exists [13]. If guarantees are present, the best known algorithm is that of [19]. We have used our tools to verify all of the examples present in [13] and [19] and our results are identical. Event graphs can also be used to represent infinite sets of inequalities which can be used to model systems with repetitive behavior, e.g., [1, 8, 11]. Our methods are not directly applicable to such event graphs.

One interesting problem discussed in [19] is a symbolic problem with a disjoint solution. The symbolic verifier of [1] cannot handle disjoint solutions and requires the event graph to have a very specific structure, and is thus not able to analyze most timing diagrams. In [19] this example (the event graph appears in Figure 2) is solved manually. Our verification tools report the following solution using .2 seconds of CPU time:

$$\{[W,X,Y,Z]:\ W \leq X \leq 50 \ \wedge\ Y \leq Z \leq 70 \ \wedge \\ (Y \geq 60 \ \vee\ (Y \leq 59 \ \wedge\ W = 50 \ \wedge\ X = 50))\}.$$

The example demonstrates an interesting aspect of our methodology. Timing diagrams usually view time as continuous, but due to our reliance on integer tuple sets and Presburger formulas, we assume time is discrete. The equation: $Y \leq 59$ could perhaps be more easily understood as $\neg(Y \geq 60)$. There exist pathological examples for which our verifier would report failure because of the need for an integer solution (e.g., examples which rely on forcing an event to take place between two other events which are constrained to be one time unit apart) but we believe this is simply an indication that the user is not working with an appropriate time scale, and this would be clearly obvious from the timing diagram itself. Furthermore, we actually need integer solutions when synchronous signals are considered, see Section 4.4.

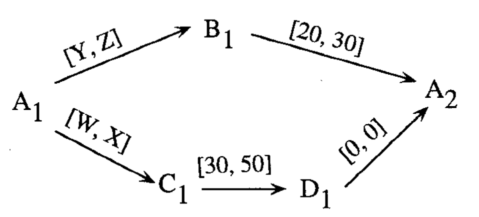


Figure 2

A simple event graph with a constraint (not shown) indicating that $80 \leq A_2 - A_1 \leq 100$. The corresponding timing diagram would be quite similar to the one in Figure 1 except that the numeric and symbolic timing information has changed, and the constraint is now from A_1 to A_2.

At this point, we must also confess that the actual solution which *tdverify* reports is not as readable as the one given above. Currently solutions are expressed as unions of sets containing no disjuncts (i.e., in sum of products form) and we manually simplify the reported formula and formally verify that no mistakes are made during simplification. We are in the process of exploring how to best report solutions for human readability.

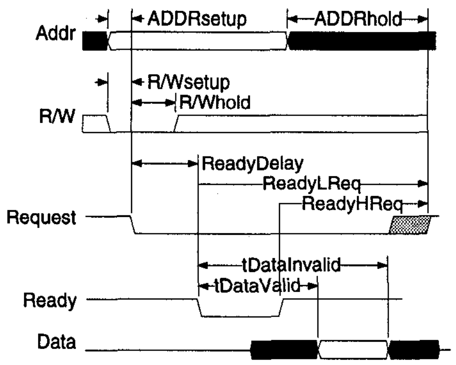
4.3 Interface timing verification

Many verification problems with respect to timing diagrams are concerned with interface verification, in which two pieces of hardware are connected, and each acts as the other's environment. The verification problem, in this context, is to decide whether or not each device satisfies the other's timing requirements. Our symbolic verifier has the capability to not only perform interface timing verification, but interface timing synthesis as well.

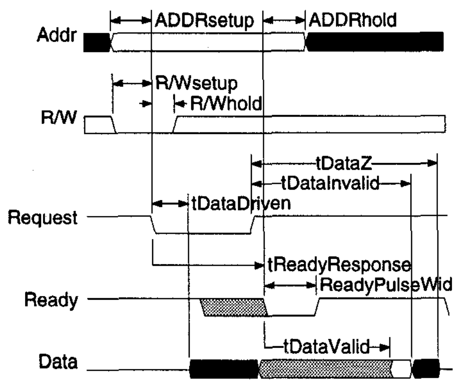
This example of interface timing verification is taken from [20] and consists of two specifications for a memory controller and a bus. Here, we consider only the read protocol. Figure 3 contains the two timing diagrams. Walkup performed the verification by creating (manually) an event graph corresponding to the composition of both timing diagrams. The event graph was analyzed and the computed maximum separations between events were compared to the required constraints. We have discovered a few minor errors in her event graph but they do not affect her results, which we were able to duplicate.

Verification takes place directly from the timing diagrams, which are combined into a single timing diagram shown in Figure 4, and then verified using our tools. At present our tool *tdmerge* requires the merged diagrams to be topologically identical. Future work is needed in this area. For example, one diagram may not contain all of the transitions present in another, and this should not prevent merging if it is clear how the two diagrams "match up." Additional concerns involve circumstances such as allowing "valid" and an actual value (e.g., "low") to result in a merging, and there are additional semantic issues that will need to be resolved in a more mature tool. Some of these issues are discussed in [2] and others have been identified in the context of timing diagram simulation [9].

Using symbolic variables (Figure 4 contains only numeric values) we can easily perform experiments which might, for example, help us to select another memory controller which would also meet our constraints. After changing the numeric values associated with



Row		Name	Formula
1	G	ADDRsetup	[30,]
2	G	R/Wsetup	[30,]
3	G	ADDRhold	[20,]
4	G	R/Whold	[20,]
5	C	tDataValid	[,100]
6	C	tDataInvalid	[140,]
7	C	ReadyDelay	[30,]
8	D	ReadyLReq	[150,180]
9	D	ReadyHReq	[10,30]



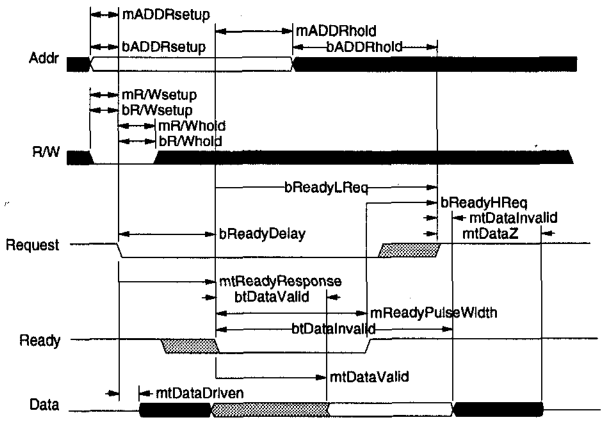
Row		Name	Formula
1	C	ADDRsetup	[25,]
2	C	R/Wsetup	[25,]
3	C	ADDRhold	[15,]
4	C	R/Whold	[15,]
5	D	tReadyResponse	[35,80]
6	D	tDataValid	[40,90]
7	G	tDataDriven	[5,]
8	G	ReadyPulseWidth	[30,60]
9	G	tDataInvalid	[0,]
10	G	tDataZ	[0,5]

Figure 3

Two timing diagrams and tables for a read-from-device protocol for a simplified bus (top) and a simplified memory controller's read operation (bottom). The nature of each relationship (constraint, guarantee, or delay) is specified to the right of the row number.

the memory (rows 5, 6, and 8) to symbolic variables ($[RRl, RRh]$, $[DVl, DVh]$, and $[PWl, PWh]$ respectively) we can ask the question: "What will the overall length of the read cycle be?" by adding an additional constraint that makes use of two new symbolic variables: $RTlower \leq Request_2 - Request_1 \leq RTupper$. Our tool reports (using .6 seconds of CPU time):

$$
\begin{gathered}
\{[RTlower, RTupper, RRl, RRh, DVl, DVh, PWl, PWh]: \\
30 \leq RRl \leq RRh \wedge \\
0 \leq DVl \leq DVh \leq 100 \wedge 0 \leq PWl \leq PWh \wedge
\end{gathered}
$$



Row		Name	Formula
1	C	mADDRsetup	[25,]
2	C	mR/Wsetup	[25,]
3	C	mADDRhold	[15,]
4	C	mR/Whold	[15,]
5	D	mtReadyResponse	[35,80]
6	D	mtDataValid	[40,90]
7	G	mtDataDriven	[5,]
8	G	mReadyPulseWidt	[30,60]
9	G	mtDataInvalid	[0,]
10	G	mtDataZ	[0,5]
11	G	bADDRsetup	[30,]
12	G	bR/Wsetup	[30,]
13	G	bADDRhold	[20,]
14	G	bR/Whold	[20,]
15	C	btDataValid	[,100]
16	C	btDataInvalid	[140,]
17	C	bReadyDelay	[30,]
18	D	bReadyLReq	[150,180]
19	D	bReadyHReq	[10,30]

Figure 4

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.

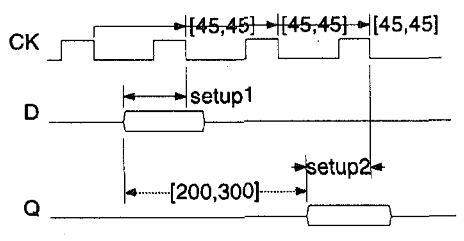
$$
\begin{gathered}
RTupper \geq 180 + RRh \wedge \\
RTupper \geq 30 + PWh + RRh \wedge \\
(RTlower \leq 10 + PWl + RRl \vee \\
(RTlower \leq 150 + RRl \wedge PWl \leq 150)).
\end{gathered}
$$

This answer reflects constraints on the type of memory controller that can be used so as to meet the bus' timing requirements (lines two and three of the result) and also provides upper (lines four and five) and lower (lines six and seven) bounds on the length of the read cycle. The disjunction involving the lower bound indicates that a tight lower bound on the length of the read cycle can only be obtained based on the value of PWl, which serves to distinguish two separate regions where different linear relationships bound the response time.

4.4 Synchronous Signals

None of the previous examples contained a clock and signals whose events were synchronous to a clock. Figure 5 contains a timing diagram we will use to describe the capabilities of our verifier with regards to synchronous signals.

Synchronous signals present many well known difficulties with regards to formal specification. Stating that two events are "one cycle apart" would not, for example, necessitate that the events are separated by a minimum of δ = *clock period.* Instead, it means that the clock edges which both events are synchronized to are consecutive. Many formal timing diagram editors provide support for specifying synchronous information in special ways.



Row		Name	Formula
1	C	constraint	[200,300]
2	G	setup1	[25,]
3	D	period	45
4	G	setup2	[25,]

Figure 5

A timing diagram containing a clock with a 45 ns. period and two signals whose events are synchronous to the clock.

As an example, consider the verification of the timing diagram specified in Figure 5. Verification fails because the guarantee on the setup time for the event on Q is expressed relative to the 8th event (the 4th falling edge) of the clock (i.e., $(CK_8 - Q_1 \geq 25)$) and the required separation between the events on D and Q is quite large (i.e., $D_1 + 200 \leq Q_1 \leq D_1 + 300$). Events which are guaranteed to be synchronous should not, however, necessarily have to specify precisely which edge of the clock they are synchronous to. Rather, we should simply state that they are synchronous to *some* edge.

At present, *tdformulize* lacks the ability to handle special synchronous constraints but, if the *tdformula* is manually edited, *tdverify* can provide meaningful results. For example, we can change the equation for the setup2 guarantee to use a symbolic variable (CC, with $CC \geq 1$) to represent some unknown number of cycles with respect to the first falling edge of the clock (i.e., $CK_2 + 45 * CC - Q_1 \geq 25$). We also need to add information which expresses something implied in the diagram, namely, that if the event on D is synchronous to the second falling edge of the clock, it occurs after the first falling edge (i.e., $D_1 \geq CK_2$) — a similar inequality will be necessary for Q as well (i.e., $Q_1 \geq CK_2 + 45 * (CC - 1)$). With these manual changes, which will eventually be performed as a part of timing diagram translation, we are able to verify the constraint and the result (generated in less than .2 seconds of CPU time):

$$\{[CC] : 6 \leq CC \leq 7\}$$

represents a determination of what clock edges would be the acceptable ones with which to synchronize Q (i.e., CK_{2+2*CC}).

There are other issues related to synchronous signals that require refining the semantics of timing diagrams. In fact, there are many other "higher level" relationships one might wish to specify using a timing diagram editor (e.g., simultaneity). Formal timing diagram editors likely will need a more expressive and explicit semantics if interface verification and synthesis are to be fully supported. We leave this for future work.

We should note that our verification methodology would break down if the clock period was expressed as a symbolic variable and we wanted to use another symbolic variable to express uncertainty regarding which clock edge we should synchronize with. In such an occasion, we would have an integer inequality which was not affine because it contained the product of two symbolic variables. As long as we avoid this situation, our verification methodology should support future extensions to better handle the semantics of timing diagrams with synchronous signals.

5 Complexity

Working with symbolic variables greatly increases the theoretical complexity of any verification methodology. The best known upper bound on the performance of an algorithm for verifying Presburger formulas is $O(2^{2^{2^n}})$ [14] and thus convention would hold that our tools may not be able to handle anything other than small trivial examples. There are many reasons why we believe this is not the case.

First, we should mention that the Omega libraries were developed for dependence analysis for advanced compiler optimizations [15]. Such optimizations rely on integer programming techniques which were thought to be too computationally expensive until the development of the Omega Test. A complete discussion of the methods used by Omega (and the newer algorithms which result in the support of Presburger formulas [17]) is outside the scope of this paper, but we should point out that a great deal of emphasis has been put on making the Omega libraries efficient and in many situations in which polynomial methods provided equivalent results, Omega has been found to have low-order polynomial worst-case time complexity [15]. One reason for this may be that Omega relies on an integer extension of Fourier-Motzkin variable elimination techniques. In our case, these techniques are especially easy to perform because all of the coefficients on our symbolic and quantified variables are always 1 or -1 unless synchronous signals are included. This is significant both for the algorithms used by Omega and for Cooper's algorithm [5].

Our Presburger formula contains only two quantifiers and they are not nested. Verifying quantifier-free Presburger formulas is NP-complete [14]. The number of quantifiers (and the structure of their nesting or alternation) can dramatically affect the complexity of performing verification. Oppen [14] has noted that even a superexponential upper bound may not be attainable if the verification algorithm puts the formula (which may grow too large) in disjunctive normal form (as does Omega). The number of disjuncts in our formula is directly related to the number of maximum or minimum functions appearing in our inequalities and this number is not likely to be extraordinarily large.

Another point in our favor is simply that timing diagrams tend to be fairly small. Modularity is used to help manage design complexity and this modularity will help with verification as well. Finally, the complexity of our approach is clearly directly related to the complexity of the interrelationships between the delays, guarantees, and constraints that are present in our specification. The amount of time spent performing symbolic timing verification is trivial compared to the amount of time that a human typically spends looking at the results. In some cases the results of verification will be used by synthesis tools, but if there is too much freedom with regards to the solution (i.e., the solution is very complex) it is likely that the results may be of limited use for synthesis. Finally, should our tools prove to be of potential use, it is quite likely that a more thorough investigation regarding verification algorithms could result in further simplifications. Even without symbolic variables, the presence of both minimums and maximums in the inequalities results in verification being NP-complete [13] and thus for many realistic verification problems exponential run times cannot be avoided.

We have performed a number of small experiments to determine if our methodology is likely to be practical for typical uses which designers might make of it. In so doing, we attempted to analyze systems that we believe are more complex than ones that would realistically be analyzed.

We analyzed the example described in Figure 1 with all of the relationships (including the constraint) expressed using symbolic variables. In this case, there were 12 symbolic variables, the verification took 3.2 seconds of CPU time and the result, consisting of the union of 16 sets, required approximately one hour of analysis for one of the authors to understand (less time would have been required if the results were reported in a more readable format).

$$\begin{tikzcd}
A_0 \arrow[rr, "{[4,10]}"] \arrow[dr, "{[1,2]}"'] & & A_1 \arrow[rr, "{[4,10]}"] \arrow[dr, "{[1,2]}"'] & & A_2 \arrow[rr, "{[4,10]}"] \arrow[dr, "{[1,2]}"'] & & A_3 \\
& B_0 \arrow[ur, "{[1,6]}"] \arrow[rr, "{[5,20]}"'] & & B_1 \arrow[ur, "{[1,6]}"] \arrow[rr, "{[5,20]}"'] & & B_2 \arrow[ur, "{[1,6]}"] &
\end{tikzcd}$$

N	3	4	5	6	7	8	9	10
seconds	.16	.37	.93	2.6	7.2	18.2	68	703

Figure 6

Lattice structured event graph out to A_3. The verifier is used to establish an upper bound on A_N - $A_{(N-1)}$ which for $N = 3$ is 24. There is one symbolic variable and the underlying equations have $2N - 1$ maximizations which result in $2^{(2N-1)}$ disjuncts which Omega must analyze. The table reports CPU times in seconds for various Ns.

We constructed a timing diagram with an event graph structure similar to a lattice, with a number of events which could be easily parameterized. Figure 6 contains a table showing execution times as a function of lattice size. Clearly there are limits to our brute force approach when maximizations and minimizations result in an exponential number of terms which need to be analyzed. Algorithms for analyzing Presburger formulas which avoid the need to put everything in disjunctive normal form exist (see [5]) but it is not known if they would perform any better than Omega. Our belief is that there exist a large class of timing diagrams which do not contain large numbers of maximizations (or minimizations) for which symbolic verification would be quite valuable.

6 Conclusion

We have developed a set of tools with unique capabilities we believe designers will find especially valuable. Our tools allow efficient verification directly from the timing diagram, and yet also support more advanced analysis. This is the case due to the general framework provided by the Omega libraries, giving us the ability to include arbitrary equations and inequalities. Thus, one can specify many "higher level" constraints that may not be directly visible in a timing diagram, such as correlated delay values [6, 7]. For example, if we have two delays in a timing diagram caused by the same component then these delays may be known to track each other (i.e., the two delays are constrained to be in a certain range, $B_1 = A_1 + \delta_1$ with $10 \leq \delta_1 \leq 20$ and $C_1 = A_1 + \delta_2$ with $10 \leq \delta_2 \leq 20$, but their difference must be small, $-2 \leq \delta_1 - \delta_2 \leq 2$).

We believe symbolic timing verification has yet to be seen by the synthesis community as an integral part of the synthesis process. It is our hope that from this example, others recognize the inherent potential that software such as the Omega libraries provide. Our future work involves bringing higher levels of abstraction to the verification process. We anticipate having to address more complicated semantics issues (e.g., improvements to *tdformulize* and *tdmerge*) as well as improve our underlying methodology and the reporting of results (e.g., improvements to *tdverify*).

REFERENCES

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

[2] BORRIELLO, G. *A New Interface Specification Methodology and its Application to Transducer Synthesis.* PhD thesis, University of California at Berkeley, 1988.

[3] BORRIELLO, G. Formalized timing diagrams. In *3rd European Conference on Design Automation (EDAC)* (Mar 1992).

[4] BRZOZOWSKI, J. A., GAHLINGER, T., AND MAVADDAT, F. Consistency and satisfiability of waveform timing specifications. In *Networks* (1991), vol. 21.

[5] COOPER, D. C. Theorem proving in arithmetic with multiplication. In *Machine Intelligence 7* (1972).

[6] GIRODIAS, P., AND CERNY, E. Interface timing verification with delay correlations using constraint logic programming. In *European Design and Test Conference* (March 1997).

[7] GLADSTONE, B. Accurate timing analysis holds the key to performance in today's system designs. *EDN* (Sept. 1993).

[8] HULGAARD, H., BURNS, S. M., AMON, T., AND BORRIELLO, G. An algorithm for exact bounds on the time separation of events in concurrent systems. *IEEE Transactions on Computers* (Nov. 1995).

[9] KHORDOC, K., CERNY, E., AND DUFRESNE, M. Modeling and execution of timing diagrams with optional and multimatch events. In *Proc. 2nd ACM Workshop on Timing Issues in the Specification and Synthesis of Digital Systems* (1992).

[10] KHORDOC, K., ET AL. Integrating behavior and timing in executable specifications. In *Proceedings of the Conference on Computer Hardware Description Languages and their Applications (CHDL)* (Apr. 1993).

[11] MARTELLO, A. R., AND LEVITAN, S. P. Temporal analysis of time bounded digital systems. In *IFIP WG10.2 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)* (May 1993).

[12] MARTELLO, A. R., LEVITAN, S. P., AND CHIARULLI, D. M. Timing verification using HDTV. In *Proc. ACM/IEEE Design Automation Conference (DAC)* (1990).

[13] MCMILLAN, K. L., AND DILL, D. L. Algorithms for interface timing verification. In *Proc. International Conf. Computer Design (ICCD)* (October 1992).

[14] OPPEN, D. A $2^{2^{2^{pn}}}$ upper bound on the complexity of presburger arithmetic. In *Journal of Computer and System Sciences* (July 1978).

[15] PUGH, W. A practical algorithm for exact array dependence analysis. In *Communications of the ACM* (Aug. 1992).

[16] PUGH, W., ET AL. The omega project. In *URL: http://www.cs.umd.edu/projects/omega.*

[17] PUGH, W., AND WONNACOTT, D. An exact method for the analysis of value-based array data dependences. In *Proc. 6th Workshop on Programming Languages and Compilers for Parallel Computing* (Aug. 1993).

[18] RONY, P. Interface fundamentals: Timing diagram conventions. In *Computer Design* (January 1980).

[19] WALKUP, E., AND BORRIELLO, G. Interface timing verification with application to synthesis. In *Proc. ACM/IEEE Design Automation Conference (DAC)* (June 1994).

[20] WALKUP, E. A. *Optimization of Linear Max-Plus Systems with Application to Timing Analysis.* PhD thesis, University of Washington, 1995.