An Approach to Symbolic Timing Verification

Tod Amon and Gaetano Borriello
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

Symbolic timing verification is a powerful extension to traditional constraint checking that allows delays and constraints to be expressed as symbolic variables. In this paper, we present an approach to symbolic timing verification using constraint logic programming techniques. The techniques are quite powerful in that they yield not only simple bounds on delays but also relate the delays in linear inequalities so that tradeoffs are apparent. We model circuits as communicating processes and our current implementation can verify a large class of mixed synchronous and asynchronous specifications. The utility of the approach is illustrated with some examples.

I. Introduction

Symbolic timing verification can be used to answer many questions about a design specification and implementation. When circuit delays are known, a symbolic verifier can provide a “yes/no” answer as to whether or not a particular implementation meets all the timing constraints in the specification (the fundamental question for non-symbolic verifiers).

More generally, however, using variables for delays can provide an answer about the range of values assignable to that variable that will still meet the constraints. This can be valuable information for a synthesis tool that must decide how to implement that particular function. When many delays are represented by variables, the answer may be a set of linear inequalities constraining the variables. These relations provide synthesis tools with information about tradeoffs between circuit delays and how implementation choices for circuit functions may affect each other.

The utility of symbolic timing verification can be further extended if we consider symbolic timing constraints. In this case, the verification tool serves as an analysis tool that can determine how circuit delays relate to the symbolic constraints. For example, if we wish to determine the maximum throughput of a circuit, a variable can be used to express a throughput constraint and the verifier can determine its range of values given circuit delays and other constraints.

It is these uses of symbolic timing verification, determining delay flexibility in synthesis and how it affects the resulting circuit architecture, that motivates this work.

II. Verification model

We model our behavioral specifications using a bipartite directed graph whose two node types are: events, which serve as reference points for timing relationships; and operations, which encapsulate different ways to interconnect events. Events can represent changes in logic level on circuit wires (from a structural perspective) or data and control flow in a hardware description language (from a behavioral perspective). Operations represent circuit functionality by taking events as inputs and determining which event outputs to generate. Events and operations are connected in a bipartite graph to form a single process loop with one thread of control. The representation is formally based on a restricted form of operation/event graphs (OEgraphs) [1, 2].

Timing constraints are expressed using the notation: “constraint(event1, event2, {≤, ≥, =}, {integer, integer variable})” and specify minimum or maximum separation times between two events. Every timing constraint can alternatively be expressed as an inequality (or a conjunction/disjunction of inequalities) involving operations’ propagation delays (expressed as values or variables). We present an example of some timing constraints and their equivalent inequalities in Figure 2.

III. Verification methodology

Verification could be thought of as a two-step process: translate each timing constraint into a non-linear inequality and then simultaneously solve the set of inequalities to find a solution. Our methodology combines these two steps into a single integrated approach.

We use a constraint transformation technique which reduces timing constraints to other “simpler” constraints and zero or more additional linear inequalities. These inequalities are immediately added to the solution using an incremental linear programming algorithm. If the set of inequalities is not solvable (e.g., X > 10, Y > X, Y < 5) this is detected and verification fails. The transformation strategy essentially consists of a list of rules which specify how to decompose constraints.

To handle delay operations, only two rules are needed. The first rule for “constraint(e1, e2, ≤, X)” applies when e1 is directly connected to e2 via a delay operation, and requires that δmax e1e2 be less than or equal to X. The second rule applies when e1 is not directly connected to e2
Now consider \((\text{constraint}(e_1, i \leq 2, 20))\) with respect to the process shown in Figure 2, i.e., \(e \rightarrow g \rightarrow i\). We see that it is trivially solved using these two rules. The second rule is applied first and results in the inequality \(T + \delta_{\max}g \leq 20\) and the new constraint \(\text{constraint}(e, g, \leq, T)\). This new constraint is handled by the first rule and results in \(\delta_{\max}e \leq T\). If the actual values of \(\delta_{\max}g\) and \(\delta_{\max}i\) are known, we can report "yes" or "no", otherwise we report our answer symbolically by removing internal variable \(T\) (i.e., \(\delta_{\max}e \leq T \leq 20 - \delta_{\max}i\)) yielding: \(\delta_{\max}e + \delta_{\max}i \leq 20\).

The rules and verification methodology have been implemented in a CLP(R) program. CLP(R) is a constraint logic programming language which provides a framework to experiment with this approach to symbolic timing verification ([5] contains an excellent summary of these languages). CLP(R) incorporates an incremental version of the standard simplex linear programming algorithm and a goal based programming language akin to Prolog. Its use has enabled us to focus our attention on the verification rules instead of the symbolic linear programming, unification, and search/backtracking aspects of the methodology. It is possible that a more efficient version of the verifier could be constructed but the version of CLP(R) we are using [6] has proven quite adequate.

The verification tool takes as input a description of the process and a list of constraints to be verified. Additional inequalities can also be supplied to further constrain the variables. The verifier currently consists of over 2500 lines of CLP(R) code implementing well over one hundred transformation rules. A large number of rules are needed to account for: the quality of the verification tool (i.e., its ability to detect and correctly process complex behaviors), the different types of operations (and the resultant behaviors which they allow), and the different types of constraints and their semantics (we have simplified the constraint representation for this paper).

Most of the rules are much more complicated than the simple rules used for delay operations and there are a number of subtleties that space constraints prohibit us from describing in detail. Some of the complexities arise due to the formal semantics of our constraints (e.g., what if \(e_1\) and \(e_2\) occur at the same time?), others arise due to the inherent complexity of the constraint with respect to the functional specification (e.g., a constraint from an event in one side of a split to an event on the other side, or constraints into and out of loops).

In some cases, multiple rules apply for a given situation. This is the case for \(\text{constraint}(e_1, e_2, \geq, X)\) where \(e_2\) is the output of a join. Every join serves to bring together two parallel paths divided by a matching split operation. The choice of transformation rules is very dependent upon whether or not \(e_1\) is an event inside the split (present in either the left or right parallel branch) or outside the split. For \(\text{constraint}(e_1, e_2, \geq, X)\) if \(e_2\) joins \(c_1\) and \(c_2\) and a matching split on \(e_3\) causes \(p_1\) and \(p_2\), and if \(e_1\) is outside the split/join then:

\[
\delta_{\max}e + \delta_{\max}i \leq 20.
\]
prove: constraint(e1, e3, ≥, T1),
and either:
add the new inequality T1 + T2 ≥ X,
prove: constraint(p1, c1, ≥, T2).
or, alternatively:
add the new inequality T1 + T3 ≥ X,
prove: constraint(p2, c2, ≥, T3).

Since e2 will not occur until both c1 and c2 occur, a minimum separation time may be ensured even if only one path actually satisfies the constraint. This represents an implicit choice for the verifier: it must decide which path to try and verify. If it succeeds, it does not need to look at the other path. If it fails, it must backtrack to this point and try the other.

The constraint could be expressed as an inequality involving a maximum function (e.g., see Figure 1). The verification problem is thus essentially nonlinear, although it can be reduced to a potentially exponential number of linear ones. Our methodology relies on a search paradigm — the search space being the many linear solutions to the non-linear formulation. Our approach is more efficient than the alternative generate and test methodology (i.e., convert constraints into nonlinear inequalities and then solve each expansion of the nonlinear inequalities until a solution is found or all expansions fail). In our methodology, an incremental linear programming algorithm is used. This means that solutions which fail can be detected without requiring a full expansion of the nonlinear inequalities (i.e., failure can occur even if many of the constraints have not yet been transformed). This early termination can prune large segments of the search space for which there is no solution. The incremental transformation process is also easy to understand (and extend) since it consists of a number of small rules used for different constraint semantics and different operations.

We now present results of symbolic timing verification for a single constraint: "constraint(g, a, ≤, 30)" for the process shown in Figure 1. Two different sets of propagation delays were used:

**Case 1:**
- \( \delta_{\text{min}}g = 70, \)
- \( \delta_{\text{max}}g = 50, \)
- (all other \( \delta \)s symbolic)

**Case 2:**
- \( \delta_{\text{min}}bd = 30, \)
- \( \delta_{\text{max}}bd = 50, \)
- (all other \( \delta \)s symbolic)

The solution for the first case is a linear inequality which closely resembles the constraint being proven in that the delay operations along the path from 'g' to 'a' are constrained: \( \delta_{\text{max}}g + \delta_{\text{max}}k + \delta_{\text{max}}ma \leq 30. \) This solution was obtained by noticing that 'k' will always occur after 'l' when occurring via 'g'. The second solution: \( \delta_{\text{min}}bd \geq 20 \) is less obvious. In this case, it is not possible to show that 'k' will always occur after 'l' (if it was, the constraint from 'g' to 'a' would be satisfied since the delay along that path adds up to 15 which is ≤ 30). The solution requires that 'g' does not occur too early with respect to 'l' since 'k' will end up waiting. This is accomplished by delaying event 'd' (and thus delaying 'g'). Note that many of the propagation delays specified as variables do not appear in the two solutions (e.g., the delay from 'f' to 'h'). This indicates that these delays are irrelevant with respect to the constraint being verified.

From a behavioral perspective, most circuits are expressed not as a single process but rather as a set of communicating processes. Extensions for dealing with some types of inter-process communication and also for modeling iterative behavior (e.g., loops) have been defined but are outside the scope of this paper (see [4]).

We have successfully applied our methodology to a small number of examples that involve cross-process constraints. As an example of the type of symbolic timing verification that we can currently perform, we present a moderately sized verification of a partial specification for the Intel Multibus. The specification consists of five communicating processing which contain a total of approximately 75 events, 65 operations, and 35 timing constraints to be verified.

There is a constraint that states that the arbitration process should not release the bus until the read/write process has finished (technically, this is expressed using two constraints and an inequality). This constraint could be eliminated if the read/write process sent a message to the arbitration process indicating completion, but this specification is less rigid; if an upper bound on the time taken by the read/write process to complete a transaction can be determined, the arbitration process need only wait a fixed amount of time before releasing the bus (leading to less circuitry). We have verified the Multibus specification in a variety of ways that help demonstrate the flexibility of our methodology.

First, we assigned delay values (intervals characterized by a minimum and maximum, not simply fixed values) to all of the operations in the specification. Before performing verification, we simulated the specification using a behavioral simulator [2] to validate our specification and ensure that what we specified was what we wanted. We then ran the verifier which reported "yes" — it was able to successfully prove that all of the timing constraints were satisfied with the given delay ranges. We then experimented with changing the delay values. We introduced a setup violation and the verifier failed (i.e., "no") when attempting to prove the setup constraint. We also experimented with setting the amount of time before the bus was released to too small a number, this also resulted in failure.
We then expressed the delay through the operation that holds up the release of the bus symbolically, using variables to represent the upper and lower bounds of the operation. The verifier reported "yes" along with constraints on the values of the upper and lower bounds. The minimum delay was constrained ("time to release bus > 30") via successfully determining the upper bound on the time taken in the read/write transaction with the memory and the lower bounds on the other delays incurred in the arbitration process (in other operations) before the bus was released. An upper bound was also given ("time to release bus ≤ 11870") due to a constraint which requires that the bus not be tied up for more than a few milliseconds. We then performed a similar experiment in which we fixed the release delay and represented the delay in the memory symbolically. In this case, the verifier reported a number of constraints on the delays which essentially answered the question "how slow can the memory be?" if we release the bus after some number of cycles.

As a final experiment, we expressed both delays symbolically and ran the verifier. It reported the relationship between the two; namely, that the time to release the bus is: ≤ 11870, > 170 + memory read access time, and > 150 + memory write access time. It also reported additional constraints with respect to both memory delays which cannot be too fast otherwise constraints in the read/write process are violated.

V. Conclusion

We believe that symbolic timing verification is a critical tool in the development of higher-level synthesis tools. The information obtainable from a symbolic verification process has three principal uses: (1) implementation verification, (2) obtaining constraints for synthesis, and (3) design evaluation. Implementation verification confirms that an implementation of a design and its associated delays will meet the constraints in the original specification. Synthesis tools can use symbolic delay values to determine the degree of flexibility that is available while still satisfying the timing constraints. This can lead to much more efficient use of resources in the final implementation. Design evaluation can be performed by using symbolic values in the constraints 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 variable to circuit delays.

In this paper, we have described an approach to symbolic timing verification using constraint logic programming as a framework. CLP(R) is an excellent test-bed for many of the algorithms. Its unification and backtracking features provide us with a mechanism for applying transformation rules and searching the space of possible solutions. We have made no attempt to optimize execution speed but have found it to be quite adequate using compiled CLP(R). The examples presented for Figure 1 take approximately five seconds of CPU time on a standard workstation (this includes compile time) and each Multibus example uses approximately two minutes of CPU time. Execution speed could be dramatically improved through the use of caching since the results of graph analysis are not currently reused. We believe that execution speed will scale well to larger specifications due to the fact that most constraints describe relationships that are local and interrelate a small number of symbolic delays. Some constraints can be quite complicated but it is unlikely that larger specifications will contain many such complex interrelated constraints. In any case, the complexity of our approach is dominated not by graph size, but by the complexity of the interrelationships and how they affect the solution space.

Our current directions are to generalize the operations and rules we currently employ in our system, especially with regard to inter-process communication and cross-process constraints. The use of symbolic timing verification in higher-level synthesis is also being explored with the synthesis of real-time controllers as a primary focus. We have already looked into queue sizing as a problem in synthesis [3] and the approach outlined here will be instrumental in obtaining the rate of send/receive events required as input to that algorithm.

Acknowledgments

The authors gratefully acknowledge numerous discussions with A. Borning and R. Yap regarding constraint logic programming and thank IBM for providing the new compiled version of CLP(R) we have been fortunate enough to use.

This work was supported by an NSF PYI Award (MIP-8858782) and by the DARPA/ISTO Microsystems Program under an ONR monitored contract (N00014-91-J-4041).

References