Automatic Generation of Chu Space Model Expressions for Verification

Lubomir Ivanov
Department of Computer Science, Iona College, 715 North Avenue, New Rochelle, NY 10801,
livanov@iona.edu

Abstract — We present an algorithm for automatic generation of behavioral models of non-iterated systems from a structural Verilog specification. The models are represented as Chu spaces over the set of system events, and are used for verifying system behavior. The correctness and time complexity of the presented algorithm are briefly discussed along with a small example.

I. INTRODUCTION

Accurate and efficient representation of concurrent behavior is an important field of research with numerous theoretical and practical applications not only to Computer Science and Engineering but to other scientific disciplines such as Physics, Molecular Biology, and even Philosophy. A number of interesting approaches have been proposed including Petri nets [17, 18], Mazurkiewicz traces [15, 16], series-parallel posets and pomsets [13, 14, 19, 21, 22, 23], event structures [18], and higher dimensional automata [3, 20]. The Chu space model of concurrent behavior was proposed by Vineet Gupta and Vaughan Pratt [1, 2]. The model has generated significant interest in the theoretical community [3-11], but, so far, has found a relatively limited application in practice [12, 24].

Our research efforts are aimed at developing an efficient, practical methodology for formal verification based on Chu spaces. System verification requires that a formal model of the given system be constructed from specifications (usually Verilog or VHDL). This formal model can then be used to verify the satisfaction of properties critical to the system’s operation. This paper presents an algorithm for the automatic generation of such a system model from a structural Verilog specification. The derived model is a Chu space represented as a Chu space algebra expression, which can be used by the verification algorithms.

The paper is organized as follows: We begin by introducing Chu spaces, and discussing a convenient (for verification) Chu space algebra. We then present an algorithm for automatic model generation from structural Verilog description, and briefly discuss its efficiency and correctness. We illustrate the use of the algorithm with an example, briefly outline the verification approach which uses the generated model, and describe directions for future work.

2. CHU SPACES: A BRIEF INTRODUCTION

A. Chu Spaces

Formally, a Chu space $(A, r, X)$ over an alphabet $\Sigma$ is defined as a triple consisting of a set $A$ of events, a set $X$ of states, and a function $r: A \times X \rightarrow \Sigma$. One way to view a Chu space is as a matrix where the rows are labeled by the events in $A$, the columns are labeled by the states in $X$, and the entries are values from the alphabet $\Sigma$ assigned by function $r$. $\Sigma$ is usually chosen to be the set $\{0, 1\}$ or $\{0, 1, 2\}$ as the values can be conveniently interpreted as describing different stages of occurrence of an event: $0$ as “The event has not occurred”, $1$ as “The event has currently occurred”, and $2$ as “The event has already occurred”. Semantically, there is no limit on the size of the alphabet $\Sigma$ and the interpretation of its values, but the larger the cardinality of $\Sigma$, the higher the space- and time- requirements of any algorithms which manipulate Chu spaces directly. Consider the following two Chu space examples:

\[
\begin{array}{ccc}
    e_1 & x_0 & x_1 & x_2 \\
    e_2 & 0 & 0 & 1
\end{array}
\]

Fig. 1

\[
\begin{array}{ccc}
    e_1 & x_0 & x_1 & x_2 \\
    e_2 & 0 & 0 & 1
\end{array}
\]

Fig. 2

The Chu space in Fig. 1 represents the fact that event $e_1$ precedes event $e_2$ by allowing only three states: $x_0$ (“Both events have not yet occurred”, $x_1$ (“Event $e_1$ has occurred, event $e_2$ has not”), and $x_2$ (“Both events have occurred”). The Chu space in Fig. 2, however, allows one additional state in which event $e_1$ has not occurred, but event $e_2$ has. Thus, it models the independence of $e_1$ and $e_2$. As an example of a Chu space over $3$, consider Fig. 3:

\[
\begin{array}{cccccccc}
    x_0 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 \\
    e_1 & 0 & 0 & 1 & 1 & 0 & 2 & 1 & 2 \\
    e_2 & 0 & 1 & 0 & 1 & 2 & 0 & 2 & 1 & 2
\end{array}
\]

Fig. 3

The Chu space in Fig. 3 also models the independence of events $e_1$ and $e_2$, but the events are allowed to be in transition, i.e. each event has three possible states – “before”, “during”, and “after”. Thus, the number of states is now nine. In general, a Chu space modeling the independence of $n$ events will have $\Sigma^n$ different states. Thus, working with Chu space matrices directly is extremely costly both in terms of time- and space complexity. This necessitates the development of more compact, implicit representations of Chu spaces that can be manipulated more efficiently. Using an appropriate Chu space algebra allows the representation of Chu spaces as compact expressions, which can be manipulated in low-order polynomial time and require limited storage space.

B. Chu Space Algebra

Our approach to modeling system behavior with Chu spaces is based on the notion of representing events occurring in the system as elementary Chu spaces over 2 or 3 as appropriate: to represent the system event $e_i$ we use the elementary Chu space $(1)$ $c_i = \{(e_i, x_{i0}, 0), (e_i, x_{i1}, 1)\}$ over 2 or the Chu space $(2)$ $c_i = \{(e_i, x_{i0}, 0), (e_i, x_{i1}, 1), (e_i, x_{i2}, 2)\}$ over 3. Next, we need to define a set of operations over Chu spaces to allow us to model event sequencing and independence. In [1-3] Gupta and Pratt presented a four-operation process algebra, which involves the operations concurrency, sequence, choice, and
orthocurrence. For the purposes of our non-iterated systems verification, the operations concurrence and sequence are sufficient. Thus, we adopt a simplified version of the Gupta-Pratt four-operation algebra and define concurrence and sequence as follows:

The **concurrence** (or **shuffle** or **shuffle**) operation of two Chu spaces \(A = (A, r, X)\) and \(B = (B, s, Y)\) is denoted by \(A \otimes B\) and is defined as the Chu space \((A + B, t, X \times Y)\) where \(t(a, (x, y)) = r(a, x)\) and \(t(b, (x, y)) = s(b, y)\). Concurrence is intended to model the independence of processes/systems \(A\) and \(B\).

To define the operation sequence (or concatenation) of Chu spaces \(A\) and \(B\), denoted as \(AB\) (or \(A \cdot B\)), the notion of a Chu-space needs to be enriched with subsets \(I\) and \(F\) of \(X\) containing respectively the initial and final states of the process being modeled. Then \(AB = (A + B, t, I)\), where \(Z \subseteq X \times Y\) consists of those states \((x, y)\) such that either \(x \in F_A\) or \(y \in I_B\). For the resulting Chu space \(A \cdot B\), \(I_{A \cdot B} = Z \setminus I_A \times Y\) and \(F_{A \cdot B} = Z \setminus X \times F_B\), i.e. \((x, y)\) is initial in \(A \cdot B\) when \(x\) is initial in \(A\), and final in \(A \cdot B\) when \(y\) is final in \(B\).

The process algebra allows us to represent a Chu space describing the behavior of a system as an expression in which the variables correspond to Chu spaces combined through concatenation and shuffle. Consider the circuit and its model Chu space below:

**Fig. 4**

**Fig. 5**

Let an “event” be defined as a change in the output of a gate as noted Fig. 4. Then, using the described process algebra, the Chu space in Fig. 5 can be represented as the expression \(M = (e_1 \otimes e_2) \cdot e_3\), where \(e_1\), \(e_2\), and \(e_3\) are elementary Chu spaces over 2 describing the occurrence of events \(e_1\), \(e_2\), and \(e_3\).

### 3. AUTOMATIC MODEL GENERATION

Below we present an outline of our model generation algorithm, followed by an explanation of its operation.

```java
/* -- Driver function -- */
boolean extractor() {
    Parse(Verilog_file);
    Associations();
    e = Output_node();
    string B = Bb(e);
    while (e = Output_node() != null)
        B = " @ " + Bb(e);
    write(B, Model_file);
}

/* -- Backward Behavior -- */
string Bb(string e) {
    if (sizeof(succ(e)) == 0)
        return e;
    else if (sizeof(pred(e)) == 1) {
        if (succ_associated(e, L) == 0)
            return e + "." + Bf(succ(e));
        else if (succ_associated(e, L) == 1)
            return "(" + e + "]@" + Bf(L) + "\)." + Bf(succ(e));
    }
    else  {
        string B = "(" + e;
        Iterator i = L.iterator();
        while (i.hasNext())
            B += "]@" + Bf(i.next());
        B += "]" + Bf(succ(e));
        return B;
    }
}

/* -- Forward Behavior -- */
string Bf(string e) {
    if (sizeof(pred(e)) == 0)
        return e;
    else if (sizeof(succ(e)) == 1) {
        if (pred_associated(e, L) == 0)
            return e + "." + Bb(succ(e));
        else if (pred_associated(e, L) == 1)
            return "(" + e + "@" + Bb(L) + "]" + B;
    }
    else  {
        string B1 = "(" + e;
        Iterator i = L.iterator();
        while (i.hasNext())
            B1 += "]@" + Bb(i.next());
        B1 += "]" + B + Bf(succ(e));
        return B1;
    }
}
```

The algorithm involves the following steps:

- Parsing the Verilog specification
- Determining event associations
- Generating the Chu space model expression

Parsing the Verilog specification involves reading through the list of components and interconnections, associating an “event” with each component, and extracting information about each event’s predecessors and successors. The information is recorded in
The second step involves determining event association. Two or more events are predecessor-associated if they share the same set of predecessors. Two or more events are successor-associated if they share the same set of successors. If two events share no predecessors/successors, then they are not associated. The conversion algorithm compares the predecessor set of each event with the predecessor sets of the remaining events, and lists the events with identical predecessor sets as predecessor-associated. The procedure is repeated with the successor sets to determine successor associativity. Determining event associativity takes a quadratic time in the size of the circuit.

The third step in the conversion algorithm is formulating the Chu space model expression. The idea is to generate expressions for each of the system’s independent sub-systems, and then combine these expressions using concurrence (⊗).

If the input event, e, of Bb() has multiple predecessors, then it initiates another recursive call to Bb(). If there is one other event which has the same predecessor as the input event, then the function returns the backward behavior of the predecessor concatenated with the event itself, i.e. $Bb(pred(e))e$, where $Bb(pred(e))$ computes and returns an expression for the backward behavior of that event.

Function Bb() begins by determining the number of predecessors of the input event. If the event has no predecessors, then the expression returned is the label of the event itself (base case of the recursion).

If the event has one predecessor, function Bb() must determine if there are other events in the system, which share the same predecessor with the input event. It consults the predecessor associativity list of the event, and if the list is empty, the function returns the backward behavior of the single predecessor concatenated with the event itself, i.e. $Bb(pred(e))e$, where $Bb(pred(e))$ initiates another recursive call to Bb(). If there is one other event which has the same predecessor as the input event, then the function returns the backward behavior of the predecessor concatenated with the shuffle of the input event and forward behavior of its predecessor-associated peer, i.e. $Bb(pred(e))e@Bb(L)^2$. The forward behavior is computed by recursive function Bf(), which is dual to Bb(). Finally, if there are multiple events predecessor-associated with the input event of Bb(), the function returns the backward behavior of the predecessor concatenated with the shuffle of e and the forward behaviors of all of its predecessor-associated peers.

If the input event, e, of Bb() has multiple predecessors, then the shuffle of their backward behaviors is computed and concatenated either with the event e only, or with the shuffle of e and the forward behaviors of all its predecessor-associated peers.

After each event has been examined, it is marked “done”, and removed from further consideration. Thus, in essence, each event is visited only once, which implies that the running time of this step is linearly bounded by the size of the circuit. Therefore, the overall running time of the conversion algorithm is dominated by the time needed to compute the event associations, and hence the time complexity of the conversion is $O(n^2)$. A detailed proof of the correctness of this algorithm is beyond the page limit of this paper. It suffices to say that the recursive calls have well-defined base cases, and, since each event is examined only once, that there is a finite number of recursive calls. Thus, the recursion terminates in a finite number of steps, yielding the desired expression.

1 Of course, if the system does not consist of independent sub-components, a single expression is generated.

2 L points to the head of the association list.

4. A SMALL EXAMPLE

To illustrate the operation of the conversion algorithm, we present a small example. Consider the circuit in Fig.6. We begin by labeling each component by $e_i$ as indicated in the logic diagram below:

![Fig 6](image)

The first and second steps of the algorithm parse the description and generate the table below:

<table>
<thead>
<tr>
<th>event</th>
<th>pred</th>
<th>pred-assoc.</th>
<th>succ.</th>
<th>succ-assoc.</th>
<th>done</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_1$</td>
<td>-</td>
<td>-</td>
<td>$e_8$, $e_9$</td>
<td>$e_5$</td>
<td></td>
</tr>
<tr>
<td>$e_2$</td>
<td>-</td>
<td>-</td>
<td>$e_5$, $e_7$, $e_1$, $e_2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$e_3$</td>
<td>-</td>
<td>-</td>
<td>$e_5$, $e_7$, $e_1$, $e_2$, $e_3$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$e_4$</td>
<td>$e_2$, $e_5$, $e_4$, $e_6$, $e_7$</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$e_5$</td>
<td>$e_1$, $e_5$, $e_7$, $e_10$</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$e_6$</td>
<td>$e_2$, $e_5$, $e_7$, $e_1$, $e_2$, $e_3$</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$e_7$</td>
<td>$e_1$, $e_5$, $e_7$, $e_1$, $e_2$, $e_3$, $e_6$</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$e_8$</td>
<td>$e_2$, $e_5$, $e_7$, $e_1$, $e_2$, $e_3$, $e_6$, $e_9$</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$e_9$</td>
<td>$e_2$, $e_5$, $e_7$, $e_1$, $e_2$, $e_3$, $e_6$, $e_9$, $e_10$</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$e_{10}$</td>
<td>$e_6$</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$e_{11}$</td>
<td>$e_6$</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Fig 7](image)

To form the Chu space model expression, we find an output node (i.e. node without a successor), and compute its backward behavior, Bb(). For the sake of the example, let us choose node $e_2$. Applying function Bb() to $e_2$ yields the following expression:

$$(Bb(e_2)@Bb(e_7)@Bb(e_9))@e_2@Bb(L) \quad (6)$$

This reflects the fact that $e_2$, $e_7$, and $e_9$ are predecessors of $e_2$, while $e_5$ and $e_6$ are predecessor-associated peers of $e_2$. The subsequent recursive calls to function Bb() with $e_2$, $e_7$, and $e_9$ as parameters expand expression (3) to the following:

$$(e_2@e_7@e_9)@e_2@Bb(L)@Bb(e_2)@Bb(e_7)@Bb(e_9) $$

One more level of recursion expands expression (4) to the following:

$$(e_2@e_7@e_9)@e_2@Bb(L)@Bb(e_2)@Bb(e_7)@Bb(e_9)@e_2@Bb(e_10) $$

The last recursive call to Bb($e_{11}$) expands the expression (5) to its final form:

$$(e_2@e_7@e_9)@e_2@Bb(L)@Bb(e_2)@Bb(e_7)@Bb(e_9)@e_2@Bb(e_10) $$

Since there are no other independently functioning sub-systems, expression (6) represents the overall system behavior. Each $e_i$ represents a Chu space over 2 or 3 as needed. The overall Chu space (over 2) represented by the above expression is given in Fig.8. Comparing this 39-state Chu space to expression (6) clearly demonstrates the appeal of using Chu-space expressions over raw Chu spaces in performing formal verification.
5. USING THE MODEL FOR VERIFICATION

The model generated by the algorithm in section 3 is used for performing formal verification of system properties. The verification algorithm takes a Chu space model expression and a Chu space property expression and checks the verification of the property in the context of the behavior model. The verification is performed with the help of two predicates: SS(B, P) verifies that the property P is sometimes satisfied within the behavior B, while AS(B, P) verifies that the property P is always satisfied within the behavior B. For lack of space we will only point out that the predicates use the canonical representation of the behavior and property expressions to recursively decompose the property and test the satisfaction of each sub-property and the correct sequencing or independence of the sub-properties with respect to the behavior.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we presented an algorithm for automatic generation of Chu space models of system behavior from structural Verilog specification. The generated model is used for performing formal verification of system properties. The model generation algorithm and the software based on it eliminate the necessity for users to gain in-depth familiarity with the formal theory of Chu spaces and Chu space verification. Thus, any practicing engineers with a working knowledge of the Verilog hardware description language can use the developed tools for verifying the correct behavior of the systems they are working on.

Current work is aimed at expanding the presented model generation algorithm in two directions — to allow the generation of Chu space models from behavioral Verilog specifications and to allow the generation of Chu space expressions for iterated systems. The latter task is particularly challenging, since Chu spaces do not naturally support the concept of iteration. Extending the presented model generation algorithm to other hardware description languages such as VHDL is also being considered. This should present few difficulties since the core algorithm remains the same and only the initial parsing of the system specification requires modification.

REFERENCES