Abstract: - We present an algorithm for generating Chu space models of the behaviors of arbitrarily complex non-iterated systems including those with N-type dependencies. The generated Chu space can be used for verification of system properties. The application of the model generation algorithm is illustrated with a few examples.

Key-Words: - concurrency, Chu spaces, system behavior modeling, N-type dependencies

1 Introduction
Generating accurate models of concurrent system behavior is important for most areas of Computer Science and Engineering as well for other disciplines such as Physics, Astronomy, and Molecular Biology to name but a few. 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]. In [1, 2] Gupta and Pratt proposed the use of Chu spaces to model concurrent behavior. Chu space based models have generated significant interest in the theoretical community [3-11], but, so far, have found a relatively limited application in practice [12, 24].

The main direction of our research efforts is aimed at developing an efficient, practical methodology for formal verification of hardware systems based on the concept of Chu spaces. The focal element of such a methodology is a formal model of a hardware system. The model must accurately describe the system behavior and permit the efficient automatic verification of system properties. This paper presents an algorithm for the automatic generation of such a system model expressed as a Chu-space over the set of system events.

The paper is organized as follows: We begin by introducing Chu spaces and discussing a set of operations over Chu spaces convenient for verification purposes. We then present an algorithm for automatic generation of Chu space models of system behavior and illustrate the use of the algorithm with a few small examples. Finally, we outline directions for future work.

2 Chu Spaces: A Brief Introduction
2.1 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: If \(\Sigma=2\), then ‘0’ can be interpreted as “The event has not occurred”, and ‘1’ as “The event has occurred”. If \(\Sigma=3\), ‘0’ can be interpreted as “The event has not occurred”, ‘1’ as “The event is currently occurring”, 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 larger the space- and time- requirements of any algorithms which manipulate Chu spaces directly. Consider the following two Chu space examples:

<table>
<thead>
<tr>
<th></th>
<th>(x_0)</th>
<th>(x_1)</th>
<th>(x_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(e_0)</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(e_1)</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Fig. 1

<table>
<thead>
<tr>
<th></th>
<th>(x_0)</th>
<th>(x_1)</th>
<th>(x_2)</th>
<th>(x_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(e_2)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(e_3)</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Fig. 2

The Chu space in Fig. 1 represents the fact that event \(e_0\) precedes event \(e_1\) by allowing only three states: \(x_0\) (“Both events have not yet occurred”, \(x_1\) (“Event \(e_0\) has occurred, event \(e_1\) has not”), and \(x_2\) (“Both events have occurred”). The Chu space in Fig. 2, however, allows all possible event sequences between events \(e_2\) and \(e_3\), and, thus, models the independence of \(e_2\) and \(e_3\).
As an example of a Chu space over 3, consider Fig.3:

<table>
<thead>
<tr>
<th></th>
<th>x0</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>x4</th>
<th>x5</th>
<th>x6</th>
<th>x7</th>
<th>x8</th>
</tr>
</thead>
<tbody>
<tr>
<td>e0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>e1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 3

The Chu space in Fig. 3 models the independence of events e0 and e1, but in this case 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. This is an important consideration for any automatic verification algorithm: a naïve approach relying on an exhaustive search of the state space is extremely costly both in terms of time- and space complexity. Any useful verification algorithm has to perform a more refined search (e.g. using cuts), or adopt a more compact representations of Chu spaces which can be manipulated more efficiently.\footnote{This paper will not elaborate on the specifics of our verification methodology, concentrating instead on the issues of generating the system model.}

2.2. 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 ei we use the Chu space (1) $c_i = (\{ei\}, \{(ei, x_{i0}, 0), (ei, x_{i1}, 1)\}, \{x_{i0}, x_{i1}\})$ over 2 or the Chu space (2) $c_i = (\{ei\}, \{(ei, x_{i0}, 0), (ei, x_{i1}, 1), (ei, x_{i2}, 2)\}, \{x_{i0}, x_{i1}, x_{i2}\})$ 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 concurrency and sequence are sufficient. The concurrency (or shuffle) operation of two Chu spaces $A=(A, r, X), B=(B, s, Y)$ 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 $t(b, (x, y)) = s(b, y)$. Concurrency is intended to model the independence of systems $A$ and $B$.

To define the operation sequence (or concatenation) of Chu spaces $A$ and $B$, denoted as $A \cdot B$ (or simply $AB$), 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 $A \cdot B = (A+B, r, Z)$, 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 $A \cdot B$, $I_{A \cdot B} = Z(I_A \times Y)$ and $F_{A \cdot B} = Z(X \times F_B)$, 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 $B$.

3 Automatic Model Generation

3.1 Model Generation Algorithm

The above process algebra allows us to model system behavior in terms of sequencing and independence of events occurring within a system. Each event is represented by a Chu space, and the event Chu spaces are appropriately combined with sequence and concurrency operators. As an example, consider the simple circuit and its model Chu space below:

![Fig. 4](image1)

![Fig. 5](image2)

Let an “event” be defined as a change in the output of a gate as noted Fig. 4. Then the Chu space (over 2) in Fig. 5 describes the behavior of the above circuit: events $e_0$ and $e_1$ occur independently, but $e_2$ depends on $e_0$ and $e_1$.

The following algorithm performs the automatic model generation:

```plaintext
ChuSpace Generate_Model(Specification S) {
    Table T = Parse_Specification(S);
    return Chu_Model(T);
}

ChuSpace Chu_Model(Table T) {
    ChuSpace B = new ChuSpace();
    for (each output event e in T)
        B = Shuffle(B, B(e, T));
    return B;
}

ChuSpace B(Event e, Table T) {
    if (pred(e, T).empty())
        return ChuSpace(e);
    else if (pred(e, T).size() == 1) {
        Event p = pred(e, T).first();
        return Cat(B(p, T), ChuSpace(e));
    } else {
        ChuSpace B = new ChuSpace();
        for (each event p in pred(e))
            B = Shuffle(B, B(p, T));
        return Cat(B, ChuSpace(e));
    }
}
```
The algorithm begins by parsing the system specification (currently in structural Verilog) into a table which, for each event, records its predecessors and whether or not the event is an "output" event. For each output event, the Chu space representing the behavior of that event is recursively computed by function $B(e, T)$. The Chu space corresponding to the shuffle of the behavior Chu spaces of all output events is returned. The Chu space behavior of each event is computed as follows:

- If the event has no predecessors, then its Chu space behavior is the elementary Chu space (1) or (2).
- If the event has one predecessor, then its Chu space behavior is the concatenation of the Chu space behavior of the predecessor event with the elementary Chu space of the given event.
- If the event has multiple predecessors, then its Chu space behavior is the shuffle of the Chu space behaviors of the predecessor events, concatenated with the elementary Chu space of the given event.

Functions Shuffle() and Cat() implement the definitions of concurrence (shuffle) and concatenation given in section 2.2. To understand the actual implementation, observe first that if the order of the events is fixed then each state vector (Chu space column) can be thought of as a binary (over $2^2$) or ternary (over $3^3$) expansion of a decimal number. For example if $<e_1, e_0>$ is the event vector of the Chu space in Fig. 1, then the states can be given by the set $\{0, 2, 3\}$. Thus, for efficiency both in terms of speed and storage, our implementation adopts an internal representation of a Chu space as a vector of events and a set of states represented by decimal numbers rather than binary or ternary state vectors. Next, let us examine the operations concatenation and concurrence more closely: Consider first the concatenation of the Chu spaces in Fig. 1 and Fig. 2:

```
<table>
<thead>
<tr>
<th>e0</th>
<th>x0</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>x4</th>
<th>x5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>e1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>e2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

A close examination of the concatenation operation reveals that the resultant Chu space consists of two sections: The first section consists of the state vectors of the first Chu space shifted left ("up" in Fig. 6) by the number of events in the second Chu space. In the example in Fig. 6, columns $x_0$ through $x_2$ are the state vectors of the Chu space in Fig. 1, but each has been shifted left twice (i.e. $00 \rightarrow 0000$, $10 \rightarrow 1000$, $11 \rightarrow 1100$). This corresponds to the fact that while the events in the first Chu space are taking place, the events of the second Chu space have not yet begun. The second section of the resultant Chu space corresponds to the situation where all events in the first Chu space have already occurred and the events in the second Chu space are now taking place. In other words the second section consists of the vectors of the second Chu space added to the last vector of the first section of the resultant Chu space (i.e. for the example in Fig.6, $1100+0000 \rightarrow 1100$, $1100+0001 \rightarrow 1110$, $1100+0010 \rightarrow 1111$, and $1100+0011 \rightarrow 1111$). Below is the algorithm for implementing concatenation:

```
ChuSpace Cat(ChuSpace C1, ChuSpace C2) {
    ChuSpace C = new ChuSpace();
    for (int s: C1.states()) {
        new_s=s*(C1.dim()^C2.num_events());
        // dim()=2 if C1 is over 2, 3 otherwise
        C.addState(new_s);
    }
    for (int s: C2.states()) {
        if (s != 0) C.addState(new_s+s);
    }
    C.addEvents(C1.events());
    C.addEvents(C2.events());
    return C;
}
```

Consider now the shuffle of the Chu spaces in Fig. 1 and Fig. 2:

```
<table>
<thead>
<tr>
<th>e0</th>
<th>x0</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>x4</th>
<th>x5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>e1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>e2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>e3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

The resultant Chu space in Fig 7 is obtained by pairwise concatenation of the state vectors of the two original Chu spaces. Below is the algorithm for implementing the Shuffle() function:

```
ChuSpace Shuffle(ChuSpace C1, ChuSpace C2){
    ChuSpace C = new ChuSpace();
    for (int s1: C1.states()) {
        for (int s2: C2.states()) {
            s=s1*(C1.dim()^C2.num_events());
            C.addState(s + s2);
        }
    }
    C.addEvents(C1.events());
    C.addEvents(C2.events());
    return C;
}
```
3.2 Modeling N-Type Dependencies

Consider the following simple circuit:

![Fig. 8]

In the circuit in Fig. 8, event \( e_2 \) has two predecessors – events \( e_0 \) and \( e_1 \). However, event \( e_3 \) has only event \( e_1 \) as a predecessor. In the theory of concurrency this situation is referred to as an N-type dependency: a set of events share some but not all predecessor events. N-type dependencies cannot be modeled with the shuffle operator as defined in [13, 14, 21] nor with concurrence as defined in [1-3]. This imposes significant limitations on modeling the behavior of real-world concurrent systems, as N-type dependencies are quite common. Thus, the Gupta-Pratt process algebra, as defined, is insufficient for our purposes and needs to be modified to accommodate N-type dependencies.

Due to the page limit, we shall only discuss Chu spaces over \( \mathbb{2} \). A “by-hand” analysis of the possible event occurrences in Fig. 8 reveals that the states of the system can be summarized by the Chu space:

<table>
<thead>
<tr>
<th></th>
<th>( x_0 )</th>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_3 )</th>
<th>( x_4 )</th>
<th>( x_5 )</th>
<th>( x_6 )</th>
<th>( x_7 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e_0 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( e_1 )</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( e_2 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( e_3 )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

(dec. states) 0 4 5 8 12 13 14 15

Fig. 9

To analyze the Chu space in Fig. 9, let us begin by applying our algorithm from section 3.1 to the circuit in Fig. 8. The algorithm decomposes the circuit into two sub-components: one involving events \( e_0, e_1, \) and \( e_2, \) and the other involving events \( e_1 \) and \( e_3. \) The Chu spaces below are generated as the behaviors of the output events \( e_2 \) and \( e_3 \) respectively:

<table>
<thead>
<tr>
<th>( x_0 )</th>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_3 )</th>
<th>( x_4 )</th>
<th>( x_5 )</th>
<th>( x_6 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e_0 )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>( e_1 )</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( e_2 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 10

Let us extend each of the Chu spaces in Fig. 10 and Fig. 11 to include the events in both Chu spaces:

<table>
<thead>
<tr>
<th>( x_0 )</th>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_3 )</th>
<th>( x_4 )</th>
<th>( x_5 )</th>
<th>( x_6 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e_0 )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>( e_1 )</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( e_2 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( e_3 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 12

Consider now vector \( x_2 \) of the Chu space in Fig. 13. The vector reflects the situation where events \( e_1 \) and \( e_3 \) have occurred while events \( e_0 \) and \( e_2 \) have not. This situation needs to also be reflected in the combined Chu space representing the overall system behavior: If an event has occurred within one of the sub-component Chu spaces, it must be represented as not having occurred in the overall Chu space as well. This naturally leads to the idea of combining the sub-component Chu spaces by pairwise OR-ing their respective extended states. Applying this approach to the Chu spaces in Fig. 12 and Fig. 13 produces precisely the desired Chu space in Fig. 9 as the overall Chu space model of the behavior of the combinational circuit in Fig. 8.

Notice that pairwise OR-ing of states is consistent with the definition of concurrence from section 2.2 which is intended to model the independence of sub-components sharing either no or all predecessors. Thus, we could re-implement the Shuffle() function introduced earlier purely as an OR-ing of the extended input Chu spaces. For efficiency, however, we opt to treat the case of sub-component Chu spaces sharing predecessor events separately since only then extending the input Chu spaces first is required:

```java
ChuSpace Shuffle(ChuSpace C1, ChuSpace C2)
{
    if (!C1.hasCommonEvents(C2) &&
        !C2.hasCommonEvents(C1)) {
        ChuSpace C = new ChuSpace();
        for (int s1: C1.states()) {
            for (int s2: C2.states()) {
                s = s1 * (C1.dim()^C2.num_events());
                C.addState(s + s2);
                C.addEvents(C1.events());
                C.addEvents(C2.events());
                return C;
            }
        }
    }
    else
        return Merge(C1, C2);
}
```
ChuSpace Merge(ChuSpace C1, ChuSpace C2)
{
    ChuSpace C = new ChuSpace();
    C.addEvents(C1.events());
    C.addEvents(C2.events());
    ChuStateExtend(C1, event_diff(C2, C1));
    ChuStateExtend(C2, event_diff(C1, C2));
    for (int s1: C1.states())
        for (int s2: C2.states())
            S.add(s1|s2);
    return C;
}

The new Shuffle() function considers first if the two input Chu spaces share no common events (which would indicate no common predecessors). If so, then the overall behavior Chu space is computed as before. If the input Chu spaces do share common events, function Merge() computes and returns the overall behavior Chu space. Function Merge() begins by setting the set of events of the overall output Chu space equal to the set of events shared between the two input Chu spaces. Next function Merge() proceeds to extend each of the input Chu space by adding rows of zeros for the events from the other input Chu space not present in the current one. This action is performed by function ChuStateExtend(), which makes use of the list of events returned by function event_diff(C2, C1), which in turn determines the events from Chu space C2 not in Chu space C1. Once the input Chu spaces have been extended, the states of the overall output behavior Chu space are computed as the pair-wise OR-ing of states from the two input Chu spaces.

3.3 A Small Example

As an example we shall employ our algorithm to generate the Chu space model for the behavior the circuit below:

![Fig. 14]

The circuit in Fig.14 has two N-type dependencies:
- pred(e2)={e0, e1} ≠ pred(e3)={e1}
- pred(e4)={e2} ≠ pred(e5)={e2, e3}

The algorithm begins by generating the predecessor table for all events:

<table>
<thead>
<tr>
<th>e</th>
<th>pred(e)</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>e0</td>
<td>-</td>
<td>no</td>
</tr>
<tr>
<td>e1</td>
<td>-</td>
<td>no</td>
</tr>
<tr>
<td>e2</td>
<td>e0, e1</td>
<td>no</td>
</tr>
<tr>
<td>e3</td>
<td>e1</td>
<td>no</td>
</tr>
<tr>
<td>e4</td>
<td>e2</td>
<td>yes</td>
</tr>
<tr>
<td>e5</td>
<td>e2, e3</td>
<td>yes</td>
</tr>
</tbody>
</table>

![Fig. 15]

Next, the algorithm recursively generates the behavior of the output event e4, which is the behavior of the predecessor of e4 concatenated with e4. Since the behavior of pred(e4) is given by the Chu space in Fig.10, the behavior of the output event e4 will be the Chu space below:

![Fig. 16]

Next, the behavior of output event e5 is generated recursively: The behavior of e5 is the shuffle of the behaviors of the predecessor events, e2 and e3, of event e5 concatenated with e5. The behavior of e2 is given by the Chu space in Fig. 10, whereas the behavior of e3 is given by the Chu space in Fig. 11. The shuffle of the behaviors of e2 and e3 is the Chu space in Fig. 9, and hence, the behavior of e5 is given by the Chu space below:

![Fig. 17]

Finally, to produce the Chu space model of the overall system behavior, the algorithm computes the shuffle of the Chu spaces in Fig. 16 and Fig. 17. Since the two Chu spaces have events e0, e1, and e2 in common, the shuffle is computed through the OR-ing of pairs of state from the extensions of the Chu spaces in Fig. 16 and Fig. 17. The Chu space model of the behavior of the circuit in Fig. 14 is:

![Fig. 18]

4 Conclusion

In this paper we presented an algorithm for generating Chu space models of non-iterated system behaviors. The Chu space model allows a more accurate description of the events dynamics within the system as it models not only the "before" and "after" states of the events, but takes into account
the "during" event state as well. If necessary, an even finer breakdown of the states of occurrence of the events can be trivially built into the algorithm. Additionally, our Chu space model generation algorithm overcomes the inability of traditional shuffle based approaches to model N-type event dependencies. Even though in this paper we only presented some simple combinational circuit examples, the algorithm can be applied to arbitrarily complex systems and protocols at varied levels of abstraction. Currently, our software generates the required Chu space model from a structural Verilog specification, but work is currently under way to add modeling from behavioral Verilog as well as from VHDL specifications. The generated model is used to perform Chu space based formal verification of system behavior.

The main thrust of our current research endeavors is in two directions: generating more compact representations of Chu spaces and developing modeling and verification methodology for iterated systems. In addition, we are working on improving both the functionality and the interface of our modeling and verification software to make it more usable to practitioners without in-depth knowledge of Chu spaces and formal verification methodology.

References: