New Techniques for Sequential Software Synthesis from a Polychronous Data Flow Formalism

Bijoy A. Jose, Student Member, IEEE, and Sandeep K. Shukla, Senior Member, IEEE,

Abstract—Much of the design and development of embedded software has been done manually over the years with rigorous posteriori verification steps to ensure correctness of the manually written code. Formalization of requirements to make verification and simulation test bench generation more rigorous are being practiced increasingly, but currently, there is much left to be desired in ensuring complete functional and timing correctness of safety-critical software. Polychronous data flow is a formalism used to express specification and to synthesize sequential software. Existing compilers operating on polychronous specification try to build a hierarchic tree-like intermediate form, bottom-up to relate signals in a design. We propose a Boolean theory based alternative where existing mathematical tools are utilized to generate embedded software from polychronous specification. Relative rates of occurrences of events on signals are identified and a unique top-down order of execution is built. Optimization techniques to improve synthesis time is also provided to enhance the performance of an associated software synthesis tool capable of visually capturing and simulating polychronous specification.

Index Terms—Synchronous systems, synchronous data flow, software synthesis, code generation.

1 INTRODUCTION

The majority of design and development of embedded software have continued to remain manual due to several reasons such as lack of software synthesis tools, unoptimized code generated by tools, and lack of formalisms. It is universally accepted that formal specification at a high level of abstraction allows for functional requirements to be expressed unconstrained by implementation issues. A manual or handwritten code in natural language specification already mixes implementation details and impose platform, implementation language, and architectural constraints with the code. Many optimization decisions are already made by the programmer at this lower level of abstraction. Thus, various opportunities for implementation specific optimizations are lost. A formal specification formalism, and a corresponding formal refinement based synthesis technique and tool can circumvent the above mentioned issues: First, this moves the functional correctness proof obligations to the higher abstraction level – the formal specification of the system; Second, the higher abstraction level specification yields itself to retargetting and platform specific optimizations.

The use of model driven code generation practices in safety critical systems is endorsed by a study conducted by researchers for Federal Aviation Administration and US Department of Transport-

• B. A. Jose and S. K. Shukla are with FERMAT Lab, Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, Virginia 24060, USA. E-mail: bijoy@vt.edu, shukla@vt.edu

tion [1]. It recommends the use of sophisticated model driven tools in design, code generation and verification phases to achieve higher quality of code with a lower time to market. Signal processing tools such as Simulink [2] and LabVIEW [3] provide software synthesis toolboxes in their respective visual modeling framework. But, the generated code from existing software synthesis tools are treated as handwritten code due to their lack of formal models of computation, and full Validation & Verification needs to be carried out on them which is the major part of a software design cycle.

A number of models and corresponding tools have been developed in Europe for software synthesis based on a synchronous model of computation. The synchrony hypothesis at the heart of these formalisms assumes that the time taken for a computation or communication is negligible compared to the time between two subsequent events on the same signal. Informally, a signal in such models is an infinite sequence of values (as most of these systems run ad infinitum as reactive systems). Signals could be input, output or internal to the system. Events occur on the signals when input arrives on an input signal, or output is computed on an output signal, or a value changes in an internal signal. The computation is paced by rounds called reactions. The rounds are paced by a global sequence of events, called ‘tick’ or events that sequentializes all activities in the system. Synthesis tools such as Esterel Studio [4], SCADE [5], Averest [6], etc. are based on synchronous programming languages Esterel [7], LUSTRE [8], Quartz [9] respectively. These have commonality in the way time
is modeled. The model of time is a totally-ordered set of events that indicate the global time progression. Signals have a particular values (including absence of a value) at specific points in this global time line. These time points represent logical instants. Computations are reactions to new events arriving at these logical instants.

An alternate formalism known as multi-rate or polychronous formalism exists, where the timing model is no longer global. The signals may have their own pace, which is asynchronous to the pace of other signals. The relations between the paces of the signals are expressed by ‘clock relations’. Events on one signal may or may not be related (by ‘happened before’) relationship. This may result in multiple time lines that may be asynchronous with respect to each other. In other words, the model of time here is partially ordered. One specific language with this model of time is SIGNAL [10] and its associated software synthesis tool is called Polychrony [11]. The partial order model of time expresses concurrency very well, but the concurrency may be degenerate and may actually imply a specific sequential order. In such cases, one could consider a polychronous model to express a sequential computation. The degeneracy comes from the fact that all the constraints on the partial order in the timing model actually implies that the partial order is a total order. In order to perform sequential software synthesis, one has to discover this degeneracy. The way to infer such a totally ordered time line is by identifying a specific signal (aka root clock signal) that has a simultaneous events with events on every other signal in the design. Since events on the root clock signal is linear, a reference time line is established with fixed relation to every other signal event. Given a SIGNAL model, if we require that the model be actually a degenerate one so that all its possible behaviors are sequential, and for specific input sequence the output sequence is determinate, then we can synthesize sequential software from the model. The synthesized software will have all the behaviors implied by the model, and no behavior that is not implied by the model – thus a correct by construction synthesis leading to behaviorally equivalent specification and implementation. We call the specific conditions on the model that lead to this degeneracy of the timing model as synthesis conditions.

Apart from requiring the identification of a root clock signal, a sequentially implementable polychronous design should be free of causal loops and must have a hierarchic clock relationship [12] with every signal in the design. Irrespective of the single or multi-rate formalism, sequential synthesis tools for synchronous languages check for the existence of unique behavior which provides a deterministic output.

A visual formalism for specifying polychronous specification named as Multi-Rate Instantaneous Channel Connected Data Flow (MRICDF) was proposed in [13]. Here the computation and communication are expressed with a network of actors connected based on data flow. A visual framework EmCodeSyn [14] project was developed to capture MRICDF specification and to allow users to model polychronous designs. A Boolean prime implicate based code generation technique was developed as an alternative to the clock calculus technique of determining sequential implementability in Polychrony [15]. Different version of available PI generators were integrated into EmCodeSyn to achieve practical synthesis times. An actor elimination technique (AET) was developed as an option to provide a simpler set of Boolean equations to the prime implicate generator to reduce prime implicate computation time, while obtaining the same functional PI results [16]. Optimization techniques such as AET of MRICDF specification aims to substantially reduce synthesis times, in the order of seconds, much like other synchronous software synthesis tools. The goal of EmCodeSyn project [17] is to enable visual specification, allow the design and synthesis of sequential embedded software. So our focus is on expanding the set of sequentially implementable polychronous specifications as opposed to the current admissible ones for various synthesis tools.

**Main Contributions of this work**

1) We formalize synthesis conditions for prime implicate based sequential software synthesis technique from MRICDF polychronous formalism. We show how the new software synthesis methodology is integrated into EmCodeSyn and use case studies to explain each refinement step in the methodology.

2) Extending [16], we explain Actor Elimination Technique (AET) for simplifying MRICDF networks for prime implicate computation. We provide formal proofs which show why AET does not alter the expected behavior of the implementation. Using a customized PI generator integrated with EmCodeSyn, we to evaluate the efficiency of actor elimination technique.

The paper is organized as follows. We introduce the basic concepts of polychronous formalism in Section 2. Software synthesis from MRICDF specification using the prime implicate based technique is explained in Section 3. Actor elimination technique for faster synthesis is proposed in Section 4. The experimental results with a case study illustrating our contributions is given in Section 5 and the paper is concluded in Section 6.

**2 Preliminaries on Polychronous Formalism**

Understanding synthesis conditions from polychronous formalism requires a background
on synchronous structures. We base our formalism on synchronous structures defined in [18].

Definition 1 (Signals and Events): A signal is a totally ordered set of events. An event is an occurrence of a value on a signal. For a given signal \( x \), the set of events on \( x \) is represented as \( E(x) \). The set of all events in a synchronous system is denoted by \( \Xi \). A signal contains a value and a custom data type.

Definition 2 (Synchronous structure [18]): \((\Xi, \preceq)\) is a synchronous structure if and only if \( \Xi \) is a non-empty set of events and \( \preceq \) is a preorder on \( \Xi \) such that:

\[
\forall x \in \Xi: \{ y \in \Xi \mid y \preceq x \} \text{ is finite, where}\\
\begin{align*}
    x \sim y &\iff_{\text{def}} x \prec y \land y \prec x & \text{(equivalence)} \\
    x \prec y &\iff_{\text{def}} x \prec y \land x \bowtie y & \text{(precedence)} \\
    x \preceq y &\iff_{\text{def}} x \prec y \lor y \bowtie x & \text{(partial order)}
\end{align*}
\]

Definition 3 (Instant): If the set of all events \( \Xi \) is partitioned with respect to the equivalence relation \( \sim \), equivalence classes containing synchronous events are formed. Each of these equivalence classes is an instant. Formally put, an instant set \( \Upsilon = \Xi / \sim \). Thus an instant denotes a logical instant in which all events happen synchronously.

Definition 4 (Epoch): The epoch of a signal is a possibly infinite set of instants where the signal has events. For a signal \( x \), \( I(x) \) (or \( \bar{x} \)) represents its epoch and clearly \( I(x) \subseteq \Upsilon \). An event of \( x \) (say \( e_x \)) ‘belongs to’ an instant (say \( S \)) in the set of instants \( \Upsilon \), denoted as \( e_x \models S \), where \( S \in \Upsilon \).

Observations

1) The events on a signal are totally ordered. The behavior of a signal \( x \) is the set of events on the signal or \( E(x) \). For any two events \( x_1, x_2 \in E(x) \), either \( x_1 < x_2 \) or \( x_2 < x_1 \).

2) Precedence relation between events could be due to data dependencies between their respective signals. Data dependence is a binary relation on events, and will be represented using ‘\( \preceq \)’.

3) Epoch of a signal can be related to the data and/or epoch of another signal. They can be broadly classified into three types: synchronous \( (I(x) = I(y)) \), downsampled \( (I(x) \subset I(y)) \), upsampling \( (I(x) \supset I(y)) \).

2.1 Synchronous languages and software synthesis

Synchronous programming languages are those languages where computations are reactions to arrival of events on signals. Under the synchrony assumption, the time required for communication and computation in a synchronous system is assumed to be instantaneous. The notion of instant (Defn. 3) defines the boundary for execution of statements for each reaction. In synchronous programming community, programs are represented in terms of clock relations of individual signals. Different flavors of synchronous languages such as ESTEREL, SIGNAL, or LUSTRE, have their own representation with slight variations in their models of computation.

ESTEREL is an imperative synchronous programming language for the development of complex reactive systems. There are two parts to a signal, namely the status and the value. The status denotes whether the signal is present or absent at a given instant and on presence, value provides the data contained in the signal. There are several software synthesis tools which accept ESTEREL as input language such as Esterel Studio [4] and Columbia Esterel compiler [19], [20]. There are also variations of ESTEREL language such as Quartz [9] language and its software synthesis tool Averest [6]. LUSTRE is a declarative synchronous language based on data flow model [8]. The data flow approach allows the modeling to be functional and parallel, which helps in verification and transformations to the implementation level. The LUSTRE software synthesis tool SCADE suite [5] is being used in the design of avionics systems and is mentioned in the list of avionics related software synthesis tools released by the U.S. department of Transportation [1]. SIGNAL is a declarative language that is multi-rate [10]. SIGNAL captures computation by data flow relations and by modularization of processes. In LUSTRE, a simulation global clock is related to signal clocks from the specification as opposed to SIGNAL where signals are allowed to be unrelated. Our proposed embedded code generation language follows the SIGNAL polychronous Model of Computation and is distinguished by our Boolean theory based novel synthesis technique.

2.2 Issues in sequential software synthesis from polychronous specification

For a given polychronous (or multi-clock) specification there may not exist a unique behavior or a fixed order of computation. But this is a requirement for sequential software synthesis. With additional synchronization, it maybe possible to force a unique behavior for the system and thus be made deterministic. Using synchronous structures, let us analyze such an example to understand sequential software synthesis from polychronous specification.

Fig. 1 shows three signals \( a, b, c \) with events on each of them shown in a vertical line. Events on a signal have a unique sequential order or in other words they are totally ordered (Obs. 1). The dashed lines in Fig. 1 between events \( a_1 \) and \( b_1 \), also \( a_3 \) and \( c_2 \) denote synchronous events or events that belong to the same logical instant. The data dependencies on the three signals shown in Fig. 1 has created a partial order (Defn. 2) on certain events, where they are known to synchronous or precede the other events. Yet, there is no unique time line as a partial order is not established on all events of the three signals. One possible behavior for the
system in Fig. 1 is $a_1, b_1, c_1; b_2; a_2; a_3, c_3$. Here events within an instant are separated by ‘,’ and events between instants are separated by ‘;’. Another behavior could be $a_1, b_1; c_1, b_2; a_2, c_2$. If there existed a synchronous relation between the events $a_2, b_2, c_1$, the behavior of the synchronous system would be $a_1, b_1; c_1, b_2; a_2, c_2$. There is no other possible ordering for logical instants and hence a unique time line is established. The data dependencies between the signals will decide the execution order between them within an instant. In the absence of a dependency relation between all signals, multiple orders for signal value update exists within an instant, which does not interfere with having a unique order of instants for the whole system.

![Fig. 1. A synchronous system with three signals](image)

One of the sequential implementability conditions or *synthesis conditions* for the polychronous language SIGNAL is called *endochrony*. Endochronous specifications are those for which the ordering of data value evaluations can be completely determined at the compile time, and therefore, deterministic sequential code can be generated [10]. In endochrony, sequentiality has been a prime precondition, leading to a restrictive class of synchronous models which satisfy this condition. Now let us consider an example to understand how a non-endochronous specification may not have a deterministic output. Fig. 2 shows a polychronous computational unit Spec1 with two inputs $a, b$ and an output $c$. Spec1 is a selection operation among events arriving at inputs, with $a$ having a higher priority over $b$. A possible time line with a global reference clock $T$ is shown along with a sequential implementation in pseudo-C programming language. The code tests for presence of events at input ports and decides on writing new events at $c$. The issue here is that the arrival of events within the global clock instant is unknown. During execution, the reading input port $a$ has to be timed out to move onto reading port $b$. If the event at $a$ arrives late, it might be missed. Also, if we are late to switch to reading $b$, computation on events at $b$ may not meet the deadline imposed by the global clock. This can be seen at the clock tick $T2$ in Fig. 2, where value at $b$ is sent to $c$ instead of value at $a$. So the given specification cannot give a deterministic output without knowing more about input signals. Or in other words, the global reference clock $T$ has no relation with input signals and hence cannot decide the boundaries of a logical instant for a polychronous system.

![Fig. 2. Non-endochronous and endochronous polychronous specification](image)

Now, consider the polychronous specification Spec2 in Fig. 2, where an additional signal $r$ is introduced which can tell the presence of events at $a$ or $b$ based on $true$ or $false$ valued Boolean events at $r$. Clearly these epoch relations must be true: $I(a) \subseteq I(r)$, $I(b) \subseteq I(r)$. A possible sequential implementation is shown in Fig. 2 along with a time line with $r$ as the reference clock. Now a computational round starts only on new events at $r$ and the value of signal $r$ will tell us whether to read $a$ or $b$. The boundaries of each logical instant is defined by the arrival of a new event on $r$ and will trigger new reactions on events at $a$ and $b$. The computation is now deterministic since the arrival of events, computation on them are known as a reference to an internal signal’s clock. This specification is endochronous and can determine the order of data value evaluations at compile time.

### 2.3 Boolean theory preliminaries for prime implicate computation

Our approach on finding a unique time line for a given polychronous specification involves representation of the specification in Boolean format and identification of prime implicates of a Boolean function. We represent the clock of a signal $x$ as a Boolean variable $b_x$. Here the Boolean variable contains a $true$ valued logical instants and $false$ valued logical instants, represented by $[b_x]$ and $[-b_x]$ respectively. Note that both $[b_x]$ and $[-b_x]$ denote logical instants where the signal $x$ is present. A signal cannot have $true$ or $false$ parts at the same time. So $[b_x] \land [-b_x] = false$ and also $b_x = [b_x] \lor [-b_x]$. A Boolean theory is formed by a set of boolean equations that represent the relationships between signals. Our synthesis
that can cover max terms of a function. Hence we provide their definitions as well. Maxterms in Boolean algebra are ‘logical OR’ operations on Boolean variables. Covering a max term can be understood as the sum term can be simplified into the individual max terms.

Definition 5 (Implicate): An implicate is a sum term that can cover max terms of a function.

Definition 6 (Prime Implicate): A prime implicate is a sum term that is not covered by another implicate of the function.

3 Software synthesis from MRICDF specification

We review the MRICDF specification format in Section 3.1 and software synthesis tool EmCodeSyn in Section 3.2. Later in Section 3.3, we explain our Boolean theory based novel synthesis technique to generate sequential software.

3.1 MRICDF formalism for synchronous modeling

Multi-rate Instantaneous Channel connected Data Flow or MRICDF is a synchronous data flow model consisting of actors communicating over instantaneous channel [15]. Computation to be performed is prescribed in terms of actors and communication is performed between actors through instantaneous channels. MRICDF formalism assumes computation and communication is performed instantaneously as per synchrony hypothesis. MRICDF model is specified in terms of its actor primitives shown in Fig. 3. ⊥ is used to denote absence of an event.

1) Function (F): A Function actor performs any user specified computation which is contained within an instant. For a Function actor \( F(n, m) \), there exists \( n \) input ports and \( m \) output ports, and it is fired only when all \( n \) input ports have events for a particular instant. The computation will result in events at all \( m \) output ports of the actor. The input and output signals will have events always for the same instant or in other words, they have the same epoch. For a Function actor \( F(n, m) \) with input ports \( i_j \) where \( j = 1..n \), and output ports \( o_k \) where \( k = 1..m \), \( \hat{i}_1 = \hat{i}_2 = .. = \hat{i}_n = \hat{o}_1 = \hat{o}_2 = .. = \hat{o}_m \).

The data type of Function actor can be integer, Boolean, float etc. In Fig. 3, a two input-single output Function actor performing an add operation on a possible set of inputs is shown.

2) Buffer (B): A Buffer actor stores the data present in the event arriving at its input port and produces an output event which contains the previously stored data. At the start of computation an initial value has to be provided for the Buffer actor. This is a single-input single-output actor without data type constraints, and it can be repeated to create a buffer of different sizes. The input and output events occur within the same instant and hence the epoch constraint associated with Buffer actor is \( i = \hat{o} \). In Fig. 3, a Buffer actor of integer type with an initial value 8 is shown.

3) Sampler (S): A Sampler actor samples the input signal arriving at the first input port when a true valued event arrives at the second input port. The data types of the first input port and the only output port are the same, and without restrictions. The second input port has to be of Boolean type. The epoch of the output signal will be the intersection of epochs of the two input ports, as input events on both of them are required for the actor to fire. The true valued events of a signal \( i \) can be represented separately as [\( i \)]. Hence for the Sampler actor, \( \hat{a} = \hat{i}_1 \cap \hat{i}_2 \).

4) Merge (M): A Merge actor merges the input events on both input ports and generates output events at the only output port. The event occurring at the first input port is passed onto the output port when both input ports have events on them within an instant. The data type of all three ports have to be the same. The output epoch is the union of the two input epochs, i.e. \( \hat{o} = \hat{i}_1 \cup \hat{i}_2 \).

The epoch relations and Boolean equations representing each MRICDF primitive actor is summarized in Table 1. More detailed examples on MRICDF modeling is given in [13]. In this paper, we focus on the software synthesis from MRICDF and the optimizations done to reduce synthesis time.

3.2 EmCodeSyn : Code synthesis tool from MRICDF specification

EmCodeSyn is a visual framework for specifying MRICDF networks where a Qt based front end was provided to pictorially represent actors [14]. The design flow methodology of EmCodeSyn is shown in Fig. 4. Our first synchronous data flow synthesis strategy followed the global time line model where
the testing of all input signals to an MRICDF network was allowed as described earlier in Fig. 2. Actors are fired in a fixed order decided by a scheduling algorithm which evaluates firing conditions based on input events. A continuous ‘while (1)’ loop in the generated Main file would read input events and trigger actor computation which are defined in a Function definition file and Header file.

With our new prime implicate based software synthesis technique, the methodology was altered to add a prime implicate based follower set generation path illustrated by the dash-lined boxes in Fig. 4. The new prime implicate based code generation technique evaluates MRICDF networks in the Boolean domain to check if a deterministic code synthesis is possible. In Epoch Analysis stage, after causality analysis, epoch equations are converted to Boolean equations to perform implementability checks using a prime implicate generator. The EmCodeSyn synthesis conditions: master trigger identification and follower set generation are explained in later sections. If the implementation criteria are fulfilled, code generation step is performed where the C files are generated. On MRICDF networks which fail to meet synthesis conditions exogenous constraints may be given to force a deterministic output and redo the code generation process.

3.3 Prime implicate based code generation strategy

The sequential implementability of a synchronous structure was discussed informally in Section 2.2, where the need for a unique order of execution was observed. In polychrony, synthesis condition (Defn. 7) is to identify an acyclic, hierarchical clock tree where the root node represents the signal with highest clock or epoch [12].

Definition 7 (Polychrony-Sequential synthesis [12]): A SIGNAL program with a process P is compilable if it is well-clocked and acyclic.

In this work, we formalize software synthesis conditions from MRICDF specifications. Akin to hierarchical clock tree of Polychrony, in an acyclic MRICDF network, the signal with highest epoch has to be identified and epoch relations between all signals in the network must be found. Here we discuss the three conditions for sequential software synthesis from MRICDF specification: absence of causal loops, existence of a master trigger signal and a follower set.

3.3.1 Causality of MRICDF networks

The absence of causal loops has to be verified for declaring an MRICDF network acyclic. First we define topological loops and buffered loops in an MRICDF network.

Definition 8 (Topological Loop): In a polychronous specification containing a sequence of actors \( a_1, a_2, ..., a_n \), if one output port of \( a_n \) is connected to one input port of \( a_1 \) and for actors \( a_1, a_2, ..., a_{n-1} \), if one output port of the \( a_i \)th actor is connected to one input port of the \( a_{i+1} \)th actor, then the connecting signals between these ports form a topological loop.

Definition 9 (Buffered Loop): A topological loop where at least one of the actors in the loop is a storage element (Buffer actor) is called a buffered loop.

The presence of a Buffer actor in a topological loop would break the causal loop condition in an MRICDF network for a particular instant. So we define a subclass of Buffer-free loops known as apparent causal loops.

Definition 10 (Apparent Causal Loop): Consider a polychronous specification with a topological loop containing signals \( x_1, x_2, ..., x_n \) having event sets \( E(x_1), E(x_2), ..., E(x_n) \) respectively. The topological loop is an apparent causal loop, if and only if, it is not a buffered loop and there exists \( S \in \Sigma \), \( \forall_{i=1}^{n}, \exists e \in E(x_i) \land e \in S \).

EmCodeSyn looks for buffer-free loops initially and evaluate Boolean equations to check if a causal loop can be constructed. Accordingly, an MRICDF network is accepted or rejected.

3.3.2 Existence of Master Trigger signal

A sequentially implementable polychronous model is endochronous in nature. Endochronous specifications

<table>
<thead>
<tr>
<th>MRICDF Actor</th>
<th>Epoch relation</th>
<th>Boolean Equations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function ( F(2,1) )</td>
<td>( \bar{a} = b = c )</td>
<td>( b_a = b_c, b_b = b_a )</td>
</tr>
<tr>
<td>Buffer ( B )</td>
<td>( \bar{a} = b )</td>
<td>( b_a = b_b )</td>
</tr>
<tr>
<td>Sampler ( S )</td>
<td>( \bar{a} = b \cap \bar{e} )</td>
<td>( b_a = b_b \land { b_c }, b_b = { b_c } \lor { \lnot b_c }, { b_c } \land { \lnot b_c } = false )</td>
</tr>
<tr>
<td>Merge ( M )</td>
<td>( \bar{a} = b \cup \bar{c} )</td>
<td>( b_a = b_b \lor b_c )</td>
</tr>
</tbody>
</table>

TABLE 1
MRICDF actor primitives and their epoch constraints

Fig. 4. Software synthesis methodology of EmCodeSyn
have a unique time line with a totally ordered set of instants. As explained in Section 2.2, we need to identify a signal with the same instant set to act as a reference signal known as the Master trigger signal.

**Definition 11 (Master Trigger [15]):** Let $M$ be an MRICDF model. Let $\text{'t}'$ be a signal with $E(t)$ as its set of events and $\mathcal{Y}$ the set of all instants of the MRICDF network. For each $S \in \mathcal{Y}$, if there exists an event $e_i \models S$, then $\text{'t'}$ will be termed the master trigger for $M$.

To identify master trigger signal, each actor is represented by its Boolean equations as given in Table 1. The system of Boolean Equations $F$ defines a theory $\Sigma$ which is the set of all satisfying assignments for the system. A disjunctive clause $C'$ belonging to the theory $\Sigma$ is called the implicate of the theory. If a disjunctive clause $C'$ is such that it cannot be contained in any other disjunctive clause, it is a prime implicate of the theory. By definition, when a prime implicate is assigned a $false$ value, the system of Boolean equations do not hold.

We claim that the master trigger Boolean variable will have the same property in a Boolean system formed out of the MRICDF network, since the master trigger signal has events in all possible instants of the network. We add an equation requiring at least one of the Boolean variables is $true$ to remove the trivial solution (‘all Boolean variables are $false$’). Now, if master trigger Boolean variable is set to $false$, no other Boolean variable can have a $true$ value, since there exists no such instant where master trigger is absent. It is also a prime implicate Boolean variable, since there is no satisfying assignment possible with the trivial solution ruled out. So, if there exits a single positive literal among prime implicants of a Boolean system from MRICDF network, then it is the master trigger of the MRICDF network. Based on this concept, we proposed a test for master trigger.

**Theorem 1 (Test for Master Trigger signal [15]):** A signal $x$ in an MRICDF model $M$ is a master trigger, if and only if the corresponding Boolean variable $b_x$ in the system of Boolean equations $B_M$ has the property that if $b_x$ is false, every other variable is false.

The essence of the master trigger test is to use the property that the master trigger signal has to have a $true$ value for any instant in the network. So if the master trigger is set to $false$, there should not be any other valid instant. The proof for Thm. 1 has been covered in [15].

**Corollary 1 (Test for Master Trigger Clause):**

Given a set of signals (say $x, y, ..., z$) in an MRICDF model $M$, a Boolean system $B_M$ is constructed from Table 1 containing the corresponding Boolean variables (say $b_x, b_y, ..., b_z$). A given disjunctive clause $b_C$ made of positive Boolean variables of the system $B_M$ is a Master Trigger Clause, if it is a Prime Implicate of the system and has the property that if $b_C$ is false, every other Boolean variable is false.

Cor. 1 is an extension of Thm. 1 where a combination of positive literals being $false$ can set every other variable as $false$. For sequential code synthesis, the existence of a Master Trigger signal in a given MRICDF network is not a sufficient condition. The relative order of execution for rest of the signals in the MRICDF network also have to be identified. This constitutes follower set generation process where Cor. 1 is used.

The conversion of MRICDF actor to its Boolean form for master trigger identification test is shown in Fig. 5. Each primitive actor is first expressed in terms of its Boolean equations from Table 1 and a Boolean Function is formed as a conjunction of their Boolean equations. Other than the epoch relations of each actor, an additional equation which adds the clause ‘the disjunction of all Boolean variables is $true$’ is part of the Boolean Function. This is represented by equation 2 for Buffer, Sampler and equation 3 for Function. This equation requires one of the variables to be $true$ and hence the trivial solution when all Boolean variables are $false$ is not applicable anymore. Each conjunctive clause in the Boolean function $F_B$ is formed by the property $b_a = b_b$ implies ((($b_a * b_b$) + ($b_a * b_b$)). The PI generator accepts variables in Conjunctive Normal Form (CNF) with numbers representing variables. For endochronous actors Buffer and Function shown in Fig. 5, a system of 2 and 3 variables is constructed respectively, which is fed to a PI generator. The PI candidates are obtained, of which the variable $a$ is avoided since it is an output variable. For non-endochronous actor Sampler, Boolean functions are more complex. The PI candidate ($b_b + b_c$) contains multiple literals, whereas ($b_b + b_{[c]} + b_{[-c]}$) also contains negative literal. Hence there is no master trigger in the system, which validates the non-endochronous nature of the actor. To force endochronous behavior exogenous information needs to be provided as in the case of Fig. 2. The endochronization of Merge actor is discussed along with the follower set generation process in the following section.

3.3.3 Follower set generation

The follower set is a totally ordered set of epochs in an MRICDF network. Each element of follower set is a set of signals with a particular epoch. The first element of the follower set contains individual signals with the highest epoch, which are identified using the test in Thm. 1. The next element contains the individual signals or several disjunctions of signals having the next highest epoch with a defined relation with the previous element. A direct relation with signals of previous element is required, since knowing a signal epoch is lower than another signal is not sufficient to compute a signal. This lets us reject specifications
Similarly $y$ will denote instants where $c$ is present or absent based on its value. Thus a Boolean function is formed by the 8 Boolean equations representing endochronized Merge actor in Round One shown in Fig. 6. The signal epoch of $x$ and $y$ is selected as master trigger and form the first element of follower set. Now the Boolean variables $x$, $y$, $c$ are set to true to form a reduced system of Boolean equations. A Boolean function consisting of 7 variables is sent to the PI generator and several PI results are obtained. There is no requirement of a single positive literal (Cor 1) and hence a PI result $(b_i + b_j)$ is valid with $(b \lor c)$ added to the follower set. Now we have all signals barring the output signal as part of the follower set and the output signal can be computed from the signals in the follower set. So a unique order of computing signals in the MRICDF network is formed using the exogenous information provided during endochronization. The final follower set of Merge actor is $F_M = \{\{x, y\}, \{(b \lor c), (x \lor [y]), a\}\}$. This is equivalent to the clock tree shown in Fig. 6 at the bottom with $x$, $y$ as root clock and $b$, $c$ as child nodes.

**BUFFER**

<table>
<thead>
<tr>
<th>Function</th>
<th>Sampler</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a = \text{Buffer } b$</td>
<td>$a = \text{Function } b$ c</td>
</tr>
<tr>
<td>Boolean equations</td>
<td>Boolean equations</td>
</tr>
<tr>
<td>$1) b_i = b_i$</td>
<td>$1) b_i = b_i$</td>
</tr>
<tr>
<td>$2) b_i + b_j = true$</td>
<td>$3) b_i + b_j + b_k = true$</td>
</tr>
<tr>
<td>Boolean Function</td>
<td>Boolean Function</td>
</tr>
<tr>
<td>$F_S = {(b_i, b_j), {(b_i \cdot b_j)} \cdot (b_i \cdot b_m) \cdot b_k \cdot b_m$</td>
<td>$F_S = {(b_i \cdot b_j)} \cdot (b_i \cdot b_m) \cdot b_k \cdot b_m$</td>
</tr>
<tr>
<td>CNF</td>
<td>CNF</td>
</tr>
<tr>
<td>$1 \cdot b_i, 2 \cdot b_j$</td>
<td>$1 \cdot b_i, 2 \cdot b_j$</td>
</tr>
<tr>
<td>p sat 2</td>
<td>p sat 2</td>
</tr>
<tr>
<td>${1 \cdot {1 \cdot 2 } \cdot {1 \cdot 3 } \cdot {2 \cdot 3 } }$</td>
<td>${1 \cdot {1 \cdot 2 } \cdot {1 \cdot 3 } \cdot {2 \cdot 3 } }$</td>
</tr>
<tr>
<td>Prime Implicates: $(b_i, b_j)$</td>
<td>Prime Implicates: $(b_i, b_j)$</td>
</tr>
<tr>
<td>Master Trigger: $(b)$</td>
<td>Master Trigger: $(b, c)$</td>
</tr>
</tbody>
</table>

**BUFFER**

<table>
<thead>
<tr>
<th>Function</th>
<th>Sampler</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a = \text{Sampler } c$</td>
<td>$a = \text{Sampler } c$</td>
</tr>
<tr>
<td>Boolean equations</td>
<td>Boolean equations</td>
</tr>
<tr>
<td>$1) b_i = b_i$</td>
<td>$1) b_i = b_i$</td>
</tr>
<tr>
<td>$2) b_i + b_j = true$</td>
<td>$3) b_i + b_j + b_k = true$</td>
</tr>
<tr>
<td>Boolean Function</td>
<td>Boolean Function</td>
</tr>
<tr>
<td>$F_S = {(b_i, b_j), {(b_i \cdot b_j)} \cdot (b_i \cdot b_m) \cdot b_k \cdot b_m$</td>
<td>$F_S = {(b_i \cdot b_j)} \cdot (b_i \cdot b_m) \cdot b_k \cdot b_m$</td>
</tr>
<tr>
<td>CNF</td>
<td>CNF</td>
</tr>
<tr>
<td>$1 \cdot b_i, 2 \cdot b_j, 3 \cdot b_k$</td>
<td>$1 \cdot b_i, 2 \cdot b_j, 3 \cdot b_k$</td>
</tr>
<tr>
<td>p sat 3</td>
<td>p sat 3</td>
</tr>
<tr>
<td>${1 \cdot {1 \cdot 2 } \cdot {1 \cdot 3 } \cdot {1 \cdot 2 } \cdot {2 \cdot 3 } }$</td>
<td>${1 \cdot {1 \cdot 2 } \cdot {1 \cdot 3 } \cdot {1 \cdot 2 } \cdot {2 \cdot 3 } }$</td>
</tr>
<tr>
<td>Prime Implicates: $(b_i, b_j)$</td>
<td>Prime Implicates: $(b_i, b_j)$</td>
</tr>
<tr>
<td>Master Trigger: $(b, c)$</td>
<td>Master Trigger: $(b, c)$</td>
</tr>
</tbody>
</table>

**MERGE**

| a | $a = \text{merge } c$ |
| Boolean equations | Boolean equations |
| $1) b_i = b_i$ | $1) b_i = b_i$ |
| $2) b_i + b_j + b_k = true$ | $3) b_i + b_j + b_k = true$ |
| Boolean Function | Boolean Function |
| $F_M = \{(b_i \cdot b_j), \{(b_i \cdot b_j)\} \cdot (b_i \cdot b_m) \cdot b_k \cdot b_m$ | $F_M = \{(b_i \cdot b_j), \{(b_i \cdot b_j)\} \cdot (b_i \cdot b_m) \cdot b_k \cdot b_m$ |
| CNF | CNF |
| $1 \cdot b_i, 2 \cdot b_j, 3 \cdot b_k$ | $1 \cdot b_i, 2 \cdot b_j, 3 \cdot b_k$ |
| p sat 3 | p sat 3 |
| $\{1 \cdot \{1 \cdot 2 \} \cdot \{1 \cdot 3 \} \cdot \{1 \cdot 2 \} \cdot \{2 \cdot 3 \} \}$ | $\{1 \cdot \{1 \cdot 2 \} \cdot \{1 \cdot 3 \} \cdot \{1 \cdot 2 \} \cdot \{2 \cdot 3 \} \}$ |
| Prime Implicates: $(b_i, b_j)$ | Prime Implicates: $(b_i, b_j)$ |
| Master Trigger: $(b, c)$ | Master Trigger: $(b, c)$ |

**ENDOCHRONIZED MERGE**

| a | $a = \text{merge } c$ |
| Boolean equations | Boolean equations |
| $1) b_i = b_i$ | $1) b_i = b_i$ |
| $2) b_i + b_j + b_k = true$ | $3) b_i + b_j + b_k = true$ |
| Boolean Function | Boolean Function |
| $F_M = \{(b_i \cdot b_j), \{(b_i \cdot b_j)\} \cdot (b_i \cdot b_m) \cdot b_k \cdot b_m$ | $F_M = \{(b_i \cdot b_j), \{(b_i \cdot b_j)\} \cdot (b_i \cdot b_m) \cdot b_k \cdot b_m$ |
| CNF | CNF |
| $1 \cdot b_i, 2 \cdot b_j, 3 \cdot b_k$ | $1 \cdot b_i, 2 \cdot b_j, 3 \cdot b_k$ |
| p sat 3 | p sat 3 |
| $\{1 \cdot \{1 \cdot 2 \} \cdot \{1 \cdot 3 \} \cdot \{1 \cdot 2 \} \cdot \{2 \cdot 3 \} \}$ | $\{1 \cdot \{1 \cdot 2 \} \cdot \{1 \cdot 3 \} \cdot \{1 \cdot 2 \} \cdot \{2 \cdot 3 \} \}$ |
| Prime Implicates: $(b_i, b_j)$ | Prime Implicates: $(b_i, b_j)$ |
| Master Trigger: $(b, c)$ | Master Trigger: $(b, c)$ |

**Fig. 6. Endochronized Merge actor with its follower set**

Now we summarize the sequential synthesis conditions for MRICDF networks. EmCodeSyn will implement using prime implicate based synthesis technique specifications that satisfy the following property.

**Definition 13 (MRICDF - sequential synthesis):**

Sequential synthesis of MRICDF specification requires that, i) there exists no apparent causal loop in the MRICDF network (Defn 10); ii) a master trigger signal can be found by applying the master trigger identification procedure (Thm 11) on signals in the network; and iii) a follower set can be generated with no unique order of execution or in SIGNAL terminology those with unconnected clock trees.

**Definition 12 (Follower set):** Given an MRICDF model $M$ and its Boolean system of equations $B_M$, let the follower set of $M$ be $F_M$. The first element of $F_S$ contains signals satisfying Thm. 1 for $B_M$ and are data dependent only on signals that are part of $F_S$. The Boolean system $B_M$ to find each follower set element $F_S$ is generated by setting Boolean signals in $F_S$ to true in the system $B_M(i-1)$. The successive elements of follower set $F_S$, contain signals satisfying Cor. 1 and have an explicit Boolean equation in $B_M$ containing only signals that are part of $F_S(i-1)$.

Endochronous actors Buffer and Function have signals of equal epoch and hence their follower sets have only a single element: $\{(a, b)\}$ and $\{(b, c, d)\}$ respectively. The non-endochronous Merge when put to the master trigger identification test returns only one PI candidate which is not an output signal $((b_i + b_j)$, see Fig. 6). But this candidate is not a single positive literal and hence is not a master trigger signal. The non-endochronous actor needs additional exogenous information to force endochrony. The endochronization process common to any MRICDF design is demonstrated for Merge actor in Fig. 6. A root clock or a master trigger epoch is provided from external environment and a fixed number of input signals are constructed with this epoch. The number of new input signals is equal to the number of input signals to the MRICDF design. The Merge actor having two inputs $b$ and $c$ are provided signals $x$ and $y$ respectively which define the instants where they are present or absent. $[x]$ and $[-x]$ denote instants where $x$ is true and false respectively. $[x]$ and $[-x]$ also denote instants where the signal $b$ is present or absent respectively.
according to Defn. 12 which contains all signals in the network.

Working with large designs, it is observed that PI computation time increased with number of Boolean variables. Bringing PI computation times to lower values to make EmCodeSyn software synthesis practical is important. Hence we embarked on optimization techniques to reduce PI generation time which is the major contributing factor in synthesis time.

4 Actor Elimination Technique for Faster Follower Set Computation

An optimization technique for faster follower set computation could be reducing a given MRICDF network to another with a smaller set of Boolean equations. The restriction here is that all properties pertaining to follower set computation should remain intact. Since follower set is a set of signals according to their epochs, data values are not taken into account. To set ground rules on techniques that do not affect the follower set of MRICDF networks, we define the Master Trigger Equivalence of two MRICDF networks.

Definition 14 (Master Trigger Equivalence): If an MRICDF network \( M_i \) with master trigger signal \( x_i \) is reduced to another MRICDF network \( M_j \) with master trigger signal \( x_j \) using any optimization technique, then the two MRICDF networks are said to be Master Trigger Equivalent (MTE) if and only if the epoch of their master trigger signals are the same (\( \hat{x}_1 = \hat{x}_2 \)).

In each instant, a master trigger signal must have an event occurrence according to Thm. 1. So there is a close relation between a master trigger and instant set of a network. We now claim that it is both necessary and sufficient to check whether the instant set of two MRICDF networks are the same to check for MTE.

Theorem 2 (Preservation of Master Trigger Equivalence): The elimination of any signal from an MRICDF network \( M_1 \) with a master trigger \( x_1 \) results in a new MRICDF network \( M_2 \) with master trigger \( x_2 \). They are said to be Master Trigger Equivalent if and only if, their instant sets are the same.

From Defn. 14, we know that two MRICDF networks \( M_1 \) and \( M_2 \) are said to be Master Trigger equivalent (MTE), if their master trigger signals are of the same epoch. In other words, if \( x_1 \) is a master trigger for the MRICDF model \( M_1 \) with an instant set \( \Upsilon_1 \) and if any optimization results in a new MRICDF model \( M_2 \) with master trigger \( x_2 \) and an instant set \( \Upsilon_2 \), we need to prove that it is necessary and sufficient to check for \( \Upsilon_1 = \Upsilon_2 \) to prove master trigger equivalence.

Proof sketch: From definition of Master Trigger in [15], a master trigger signal \( x \) has an event in each instant set \( S \in \Upsilon \). In the model \( M_1 \), \( x_1 \) has one event in each of the \( n \) instants of \( \Upsilon_1 \). Thus instant set of all other signals are subsets of the instant set of master trigger \( x_1 \).

To prove the ‘if’ case, consider an instant \( I \in \Upsilon_1 \) with two events \( e_{x_1} \) of the signal \( x_1 \) and \( e_{y_1} \) of the signal \( y_1 \). If we remove the signal \( y_1 \) from the network, a new MRICDF model \( M_2 \) is created. Since the instants of \( y_1 \) are subsets of master trigger \( x_1 \), \( \Upsilon_1 \) remains unchanged. So the master trigger \( x_2 \) of \( M_2 \) has the same set of instants as \( M_1 \) and the master trigger \( \hat{x}_2 = \hat{x}_1 \). Hence, proved that two MRICDF models are Master Trigger equivalent, if they have the same instant sets.

To prove ‘only if’ case, consider an instant \( T \in \Upsilon_1 \) with only one event \( e_{x_1} \) of the signal \( x_1 \). If signal event \( e_{x_1} \) is removed, \( \hat{x}_1 \) has changed to a new epoch, say \( \hat{x}_2 \) which fulfills the master trigger condition for the new model \( M_2 \). The new set of abstract instants \( \Upsilon_2 \) does not contain the element \( T \) anymore. Since \( \Upsilon_2 \neq \Upsilon_1 \), the new master trigger (say \( x_2 \)) is not equal to \( x_1 \). Hence, proved that two MRICDF models are not Master Trigger equivalent, if they have different instant sets.

Corollary 2 (Preservation of Prime Implicate Epoch): The elimination of any signal from an MRICDF network \( M_1 \) having prime implicate epoch \( \hat{\pi}_1 \) results in a new MRICDF network \( M_2 \) with a prime implicate epoch \( \hat{\pi}_2 \). If the MRICDF networks \( M_1 \) and \( M_2 \) have the same instant set, then their prime implicate epochs are the same.

This is an extension of Thm. 2 which states that the instant set is unaffected by actor elimination if master trigger is preserved. A master trigger is a prime implicate which is also a single positive literal. So the epoch of a master trigger and a prime implicate is the same. If the master trigger signal epochs are the same for two MTE MRICDF networks, then their prime implicate epochs are also preserved.

4.1 Actor elimination technique rules for a MRICDF networks

Our proposed Actor Elimination Technique (AET) aims to remove actors and thereby its related signals with this broad criterion in mind: actor elimination should not alter the master trigger epoch of the MRICDF network. For analysis, an MRICDF network is represented as \((V, \Sigma)\), where \( V \) is the set of variables and \( \Sigma \) is the set of all epoch equations. Two connected actors will have their own input and output variables being equated and thus forming a signal. After the removal of epoch equations \( \Sigma_{\text{elim}} \) through AET a reduced MRICDF network \((V', \Sigma')\) is obtained. Here are some pointers to be considered while performing AET.

1) Input variables cannot be eliminated since input signal can be a master trigger candidate. Given that \( V = V_I \cup V_{\text{oth}} \), where \( V_I \) is the set of input variables and \( V_{\text{oth}} \) represents the rest which can be considered for elimination.

2) AET can eliminate a signal variable \( v_i \) if there exists another variable representative of the epoch.
of \( v_i \). An eliminated epoch equation \( v_i \) becomes a part of the eliminated Boolean equation set \( \Sigma_{elim} \), where \( \Sigma = \Sigma' \cup \Sigma_{elim} \).

3) Data type of a signal does not matter while performing AET, since we are concerned with only the epoch of the signals.

Now we present actor elimination rules for each primitive actor, \( B_{elim} \), \( F_{elim} \), \( S_{elim} \) and \( M_{elim} \).

\[
\begin{align*}
V, \Sigma \cup \{y = x\}, \Sigma_{elim} & \quad B_{elim} \\
V - \{x, y\}, \Sigma \{\frac{1}{x}, \frac{1}{y}\}, \Sigma_{elim} \cup \{x = y\} \\
V, \Sigma \cup \{z = f(\vec{x})\}, \Sigma_{elim} & \quad F_{elim} \\
V - \{\Pi_i(\vec{x}), z\}, \Sigma \{\frac{1}{\Pi_i(\vec{x})}, \frac{1}{z}\}, \Sigma_{elim} \cup \{\Pi_i(\vec{x}) = z\} \\
V, \Sigma \cup \{z = x \cap [y]\}, \Sigma_{elim} & \quad S_{elim} \\
V - \{x, y, z\}, \Sigma \{\frac{1}{x}, \frac{1}{y}, \frac{1}{z}\}, \Sigma_{elim} \cup \{z = x \cap [y]\} \\
V, \Sigma \cup \{z = x \cup x\}, \Sigma_{elim} & \quad M_{elim} \\
V - \{x, z\}, \Sigma \{\frac{1}{x}, \frac{1}{z}\}, \Sigma_{elim} \cup \{z = x \cup x\}
\end{align*}
\]

4.1.1 Buffer Actor

The Buffer actor performs no epoch modifications and the input-output port signals can be replaced by a single variable in the Boolean equations. For a Buffer actor, \( V = \{x, y\} \) and \( \Sigma = \{x = y\} \).

Fig. 7(a) shows the application of AET on Buffer actor. An MRICDF network \((V, \Sigma)\) is shown, consisting of two Function actors with a Buffer in between. It can be represented as \( V = V_R \cup V_B \) and \( \Sigma = \Sigma_R \cup \Sigma_B \cup \Sigma_{BP} \), where \( V_{BP} \) represents variables of Buffer actor and \( V_R \) is the set of all other variables in the network. Let \( \Sigma_R \) represent epoch equations with only Buffer actor variables and \( \Sigma_B \) represent equations with input and output port connections of Buffer. Then \( \Sigma_R \) represents the equations represented by rest of the MRICDF network.

\[
\Sigma = \Sigma_R \cup \Sigma_B \cup \Sigma_{BP}
\]

Going by the elimination rule \( B_{elim} \), the epoch equation \( \{x = y\} \) is eliminated and thus \( \Sigma_B \) becomes a part of \( \Sigma_{elim} \). The port connections \( x, y \) are replaced with \( i_1 \), making equations \( \{i_1 \} \) irrelevant and changing \( \{y = o_1\} \) to \( \{i_1 = o_1\} \). The Buffer actor ports will have the same epoch as the eliminated variables \( x \) or \( y \) and hence the master trigger is not altered. Hence the reduced MRICDF network \((V', \Sigma')\) is MTE to the original network \((V, \Sigma)\), where \( V = V' \cup \{x, y\} \) and \( \Sigma = \Sigma' \cup \Sigma_{elim} \).

4.1.2 Function Actor

The Function actor equates all input epochs with all output epochs. Hence a single equivalent variable to represent that epoch information is sufficient for this actor type. A Function actor for elimination interfaced with three Buffer actors is shown in Fig. 7(b). The input-output signals \( x, y, \) and \( z \) will be eliminated under AET rules.

The MRICDF network in Fig. 7(b) is represented by \((V, \Sigma)\), where \( \Sigma = \Sigma_1 \cup \Sigma_P \cup \Sigma_F \). Here \( \Sigma_F \) represents the Boolean equations of the actor \( F_1 \) that will be eliminated. \( \Sigma = \Sigma_R \cup \Sigma_P \cup \Sigma_{BP} \).

After elimination, \( \Sigma_F \) is a part of \( \Sigma_{elim} \) and the epoch equations interfacing Function actor internal variables with port variables are replaced with a new epoch equation \( \{i_1 = i_2 = o_1\} \). A new reduced MRICDF network is formed \((V', \Sigma')\), \( V = V' \cup \{x, y, z\} \) and \( \Sigma' = \Sigma_R \cup \Sigma_{elim} \), where \( \Sigma' = \Sigma_R \cup \{i_1 = i_2 = o_1\} \).

The same elimination rules apply for input and output signals of Function actors with different number of input-output ports.

4.1.3 Sampler Actor

Sampler actor performs an intersection operation the input signal epochs and generates an output signal which has a lower epoch than any of the input signals. We show that replacing the output signal of Sampler actor with an input signal does not remove any instant of the MRICDF network and is MTE to the original network.

In Fig. 7(c) a Sampler actor is shown to be connected to three Buffer actors. The MRICDF network is represented by Boolean equations \( \Sigma \) and the Sampler actor Boolean equations are represented by \( \Sigma_S \), and hence \( \Sigma = \Sigma_R \cup \Sigma_P \cup \Sigma_S \). \( \Sigma = \Sigma_1 \cup \Sigma_P \cup \Sigma_S \).

\[
\Sigma = \Sigma_1 \cup \{i_1 = x\} \cup \{i_2 = y\} \cup \{z = o_1\} \cup \{z = x \cap [y]\}
\]
We consider cases where the master trigger is fixed as an input-output signal of Sampler actor \((V_5, \Sigma_S)\) or as a signal in the MRICDF network not directly connected to Sampler actor.

a. **Master Trigger signal \(\in V_S\):** If the master trigger of the MRICDF network is an input to \(S_1\) actor (say \(x\)), the epoch of \(S_1\) actor variables will be of the form \([y] \subseteq \hat{y} \subseteq \hat{x}\) and \(\hat{z} \subseteq \hat{x}\). So the Sampler epoch equation reduces from \(\hat{z} = \hat{x} \cap \hat{y}\) to \(\hat{z} = \hat{y}\). Now \(z\) can be safely eliminated by AET since it’s epoch is lesser than the master trigger. Any of the interfacing signals with the Sampler actor can be connected to the output port and the resulting MRICDF network is MTE in nature. If the master trigger is the output signal \(z\), the Sampler epoch equation is reduced to \(\hat{z} = \hat{x} = \hat{y}\). This is similar to eliminated epoch equations of Buffer and Function and hence AET will generate a MRICDF network that is MTE in nature.

b. **Master Trigger signal \(\notin V_S\):** If the master trigger of the MRICDF network is a signal unconnected to Sampler actor input-output signals (say \(y\)), we know \(\hat{x}, \hat{y}, \hat{z} \subseteq \hat{m}\). Replacing any of the Sampler internal signals \(x, y, z\) with the input interfacing variables in Boolean equations does not affect the epoch of the master trigger. Hence we retain all instants of the MRICDF network and generate reduced MRICDF networks that are MTE in nature.

Since instants of an MRICDF network remains unaffected irrespective of master trigger being a signal connected or not connected to Sampler actor, AET can be safely applied on this type of actor with the input interface signals replacing the internal signals.

### 4.1.4 Merge Actor

Merge actor performs a union operation on the epoch of input signals and generates an output signal with a greater epoch. So any Merge actor feeding on inputs from the environment could result in the creation of a signal with master trigger epoch. So master trigger equivalence cannot be guaranteed when a Merge actor is eliminated. So as a general rule, a Merge actor is not part of our actor elimination procedure.

### 4.2 Implications of actor elimination on follower set generation

A flow chart representing the computation of follower set is given in Fig. 8. To understand the design flow and application of AET, let us consider the example given in Fig. 9(a). Here the MRICDF network consists of three actors of Sampler type and one actor each of types Merge, Function and Buffer. On applying AET, the Function and Buffer actors with the Sampler actor not connected any input port is eliminated. Prime implicates of the network shown in Fig. 9(b) is evaluated and no master trigger is found. External master trigger signals \(x_0, S_1.i.1, x_1.S_1.i.2, x_2.S_2.i.3, x_3.S_2.i.4\) are applied to provide clock information for each input. On true occurrences of these clock signals values are read from input ports \(S_1.i.1, S_1.i.2, S_2.i.3, S_2.i.4\). \(b_{S_1.i.1} + b_{S_1.i.2} + b_{S_2.i.3} + b_{S_2.i.4}\) is among the prime implicates of the newly created Boolean equations. The Sampler actors \(S_1, S_2\) can only be fired on true values at ports \(S_1.i.2\) and \(S_2.i.4\). Hence the signals \([S_1.i.2], [S_2.i.4]\) are part of the third FS element along with signals that can be computed as the actors \(S_1, S_2\) fires. Fig. 9(c) shows the actors in shaded portion, which can be evaluated for scheduling with the current epoch information. On true values at input port \(S_3.i.9\), the actors \(S_3, M_1\) will be available for scheduling.

Fig. 8. Generation of follower set using Prime Implicate method

![Flow chart](image)

Fig. 9. Elimination steps and scheduling for an MRICDF network

There are concerns on whether the order of actor elimination can affect the prime implicate result. Also actor elimination rules are applied before computing
each follower set element and this requires the preserve prime implicate epoch (Cor. 1) to be preserved. Here we provide theorems and proofs to show why the order of elimination is irrelevant and why the actor elimination technique can also be applied for each follower set element computation.

**Theorem 3 (Actor elimination order):** For a given MRICDF network $M1$ with a master trigger $x_1$, the elimination of actors or signals in different orders of elimination will eventually result in an MRICDF network $M2$ with the same instant set.

The proof of this theorem is obvious, considering Thm. 2 ensures instant set is unchanged for two MRICDF networks that are MTE. For each elimination step, the instant set is the same and hence master trigger equivalence is preserved irrespective of actor elimination order.

**Theorem 4 (Follower set preservation):** For any MRICDF network (say $M1$) with follower set $F_{M1}$ and its optimized version $M2$ with follower set $F_{M2}$ obtained on applying AET, the epoch of follower set elements $F_{M1}$ and $F_{M2}$ are the same.

**Proof:** Preservation of master trigger (Thm. 2) proved that for two MTE MRICDF networks, the instant set remains the same and it preserves the master trigger signal epoch. Cor. 2 has shown that preserving prime implicate epoch is same as preserving instant set of the original MRICDF network. By Defn. 12, follower set elements have same instants as the instant set of the MRICDF network. Thus, instant set preserving actor elimination technique will ensure that follower set element epochs remain unchanged.  

### 5 Case studies and experimental results with EmCodeSyn

EmCodeSyn [14] has been reworked to a capture MRICDF designs using a Qt-based front end with prime implicate based code synthesis. The new version evaluates software synthesis conditions using the new prime implicate technique and then performs code generation. The actor elimination technique has been integrated into EmCodeSyn for shorter synthesis time. We take an MRICDF network Absolute as a case study to explain each stage of the software synthesis process.

#### 5.1 Absolute : Case study of a polychronous specification

The sequential synthesis using EmCodeSyn of polychronous specification Absolute in the programming language SIGNAL is shown in Listing 1. Without going into details of SIGNAL, we can understand the program as a composition of four parallel statements with one input signal $inval$ and one output signal $absol$. The intention of the program is to find the absolute value of the given integer input. It performs this operation by checking if the input is less than zero, denoted by the signal $neg$. Based on this sign, the input value or its multiplication by -1 is sent as output $absol$.

**Listing 1.** Endochronous SIGNAL program Absolute

```plaintext
process absolute =
(? integer inval;
! integer absolut);
| neg := true when (inval < 0 )
default false when (inval >= 0)
| mult := (inval * -1)
| absolut := (mult when neg) default
(inval when (not neg))]
where boolean neg; integer mult; end;
```

The SIGNAL program is visualized as an MRICDF specification in EmCodeSyn (Fig. 11, 10) contains 8 actors. The Function-Sampler actor combination F1-S1 and F3-S3 computes its SIGNAL equivalent $when(inval < 0)$ and $when(inval >= 0)$ respectively. The 27 variable Boolean function representing the MRICDF design was found to be endochronous with input signal $inval$ being the master trigger. This is because any instant of the MRICDF network requires an event at $inval$ input port. The second element of $F_{absol}$ would consist of signals of lesser epoch such as $(M1i4 \lor M1i5)$, where each Merge input represents the positive and negative cases of the signal $neg$. The only signal with lesser epoch than this set would be the third element representing $mult when neg$. Thus a follower set containing signals with their execution order is formed $F_{absol} = \{ inval \}, \{ (M1i4 \lor M1i5) \}, \{ M2i9 \}$. The other signals in each $F_{absol}$ element is avoided for simplicity.
Even with the huge improvement in PI generation for EmCodeSyn, larger models continue to have high synthesis time as opposed to other compilers which can synthesize software in a few seconds. To make the use of visual modeling of MRICDF practical, lower synthesis time was necessary and hence AET was developed. Fig. 11 shows the application of AET on the MRICDF network which would eliminate three Sampler actors, while retaining Function (connected to input signal) and Merge actors. The number of epoch equations is reduced from 24 to 11 on application of AET for computing master trigger element and the entire code generation process is seen to be taking around 2 seconds.

Fig. 11. Actor elimination technique applied on the example Absolute

Code generation from this endochronous MRICDF specification results in three C files: main, header and function definition. The significant parts of the the main and function definition files for Absolute are shown in Fig. 12. The main.cpp contains an iterative loop limited to the number of input events provided from the environment (max_iterations). Within this loop, the events on the master trigger inval or F1i1 is read. For a specification that requires endochronization such as Merge, the master trigger will contain multiple input signals to be read (a.x[0][iteration],a.x[1][iteration]), which indicates the arrival of events at the two input ports of the actor. Each actor input port has a Boolean variable associated with it which denotes the presence or absence of the associated signal event. Here inval events alone are read (a.x[0][iteration]) in SECTION 1 and its associated port Boolean variables F1i1,F2i6,F3i11 and M2i10 are updated. In SECTION 2 of the generated code, based on the Boolean port variables, firing conditions of each type of actor is evaluated. Actor F1 will fire on every input event at F1i1. Other actors are scheduled according to the data dependencies. Finally in the function definition file, each actor computation is specified. Standard actor definitions for each data type is put in when required along with custom Function actors such as fc1less.

Fig. 12. Generated code for Absolute

5.2 Impact of Actor Elimination Technique on code synthesis

We analyze the impact of AET on code synthesis time for a few MRICDF examples using the customized version of prime implicate generator. The MRICDF models used range from simple counters to more complex Flight Warning System (FWS) and pEHBH, a latency insensitive handshaking protocol module. More information about these examples are summarized in a technical report with references [21]. The impact of AET on synthesis time can be understood by analyzing the numbers in Table 2. Along with the number of actors in each example, the reduction achieved in terms of number of Boolean equations with and without AET is shown in columns 3 and 4. The total synthesis time for generating C files in seconds is given in columns 5 and 6. It can be observed that application of AET results in varied improvements for different examples. The type of the eliminated MRICDF actors, if any, contribute to the amount of savings obtained. Lesser the actors connected to external inputs, more the actors that are candidates for elimination. This can be observed in the reduction of Boolean equations for endochronous examples Absolute and FWS. For examples with several prime implicates, code generation cannot begin until PI generator finishes finding all possible prime implicates. Our requirement for master trigger

```cpp
void Synth::Merge(bool a,bool b,bool& c,bool d,bool e,bool& f)
{ Synth a;
  while(iteration<a.max_iterations)
  { getline(a.myfile_F1i1,line);
    //******SECTION 1******READ master trigger events & inputs
    if(a.x[0][iteration])
      // firing conditions for actors F1, F2, F3 are set
      (a.x[0][iteration],a.x[1][iteration]), which indicates the arrival of events at the two input ports of the actor. Each actor input port has a Boolean variable associated with it which denotes the presence or absence of the associated signal event. Here inval events alone are read (a.x[0][iteration]) in SECTION 1 and its associated port Boolean variables F1i1,F2i6,F3i11 and M2i10 are updated. In SECTION 2 of the generated code, based on the Boolean port variables, firing conditions of each type of actor is evaluated. Actor F1 will fire on every input event at F1i1. Other actors are scheduled according to the data dependencies. Finally in the function definition file, each actor computation is specified. Standard actor definitions for each data type is put in when required along with custom Function actors such as fc1less.

Fig. 11. Actor elimination technique applied on the example Absolute

5.2 Impact of Actor Elimination Technique on code synthesis

We analyze the impact of AET on code synthesis time for a few MRICDF examples using the customized version of prime implicate generator. The MRICDF models used range from simple counters to more complex Flight Warning System (FWS) and pEHBH, a latency insensitive handshaking protocol module. More information about these examples are summarized in a technical report with references [21]. The impact of AET on synthesis time can be understood by analyzing the numbers in Table 2. Along with the number of actors in each example, the reduction achieved in terms of number of Boolean equations with and without AET is shown in columns 3 and 4. The total synthesis time for generating C files in seconds is given in columns 5 and 6. It can be observed that application of AET results in varied improvements for different examples. The type of the eliminated MRICDF actors, if any, contribute to the amount of savings obtained. Lesser the actors connected to external inputs, more the actors that are candidates for elimination. This can be observed in the reduction of Boolean equations for endochronous examples Absolute and FWS. For examples with several prime implicates, code generation cannot begin until PI generator finishes finding all possible prime implicates. Our requirement for master trigger
```
is to find a single positive literal prime implicate. So customizing PI generator to break when a master trigger PI is identified can reduce synthesis time further. Actor elimination technique, being non-invasive will continue to provide improvement in synthesis time by simplifying the Boolean equations sent to the PI generator. The synthesis time for Polychrony compiler [11] is also in the order of seconds for these examples. To reiterate, the goal of actor elimination technique is to bring prime implicate based software synthesis time to acceptable levels, thus enable MRICDF and EmCodeSyn as an alternative means to visually model and synthesize polychronous specifications.

6 CONCLUSION

The EmCodeSyn project is being developed to provide a visual means of specifying polychronous MRICDF networks. In this work, synthesis conditions for implementing MRICDF networks as sequential C code have been formally defined. A prime implicate based software synthesis strategy for computing the individual synthesis conditions has been integrated into EmCodeSyn. High synthesis time due to an external prime implicate generator has been addressed by providing optimizations to simplify the Boolean system sent to the prime implicate generator. These optimizations have been proven to be devoid of any changes in program behavior in the final implementation. Our non-invasive actor elimination approach removes computations which do not affect the synthesis criteria and is compliant with improvements made to the external prime implicate generator. Also, code generation from the collaterals obtained from synthesis criteria checking is explained in terms of a sample MRICDF network. Our current effort includes causality analysis [22], modeling of more polychronous examples, and the integration of verification tools into EmCodeSyn.

ACKNOWLEDGMENTS

We acknowledge the contributions of Jason Prible, Lemaire Stewart, Matthew Kracht and Jasdeep Malhotra in the development of EmCodeSyn tool. We are thankful to Neil V. Murray, Erik Rosenthal and Andrew Matusiewicz from University at Albany for their work on the prime implicate generator. This research has been partially funded by Air Force Office of Scientific Research and National Science Foundation.

REFERENCES


<table>
<thead>
<tr>
<th>MRICDF network</th>
<th># of actors</th>
<th># of Boolean eq.</th>
<th>MRICDF network</th>
<th># of actors</th>
<th># of Boolean eq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Height-supervisor</td>
<td>8</td>
<td>71</td>
<td>Absolute</td>
<td>8</td>
<td>39</td>
</tr>
<tr>
<td>Absolute</td>
<td>8</td>
<td>24</td>
<td>Factorial</td>
<td>8</td>
<td>36</td>
</tr>
<tr>
<td>Factorial</td>
<td>8</td>
<td>25</td>
<td>Resettable counter</td>
<td>8</td>
<td>24</td>
</tr>
<tr>
<td>Resettable counter</td>
<td>8</td>
<td>15</td>
<td>ePHBRI</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>ePHBRI</td>
<td>8</td>
<td>32</td>
<td>Producer-consumer</td>
<td>15</td>
<td>40</td>
</tr>
<tr>
<td>Producer-consumer</td>
<td>15</td>
<td>40</td>
<td>Watchdog timer</td>
<td>16</td>
<td>60</td>
</tr>
<tr>
<td>Watchdog timer</td>
<td>16</td>
<td>60</td>
<td>Flight Warning Syst.</td>
<td>17</td>
<td>48</td>
</tr>
</tbody>
</table>

TABLE 2

Impact of actor elimination technique

<table>
<thead>
<tr>
<th>MRICDF network</th>
<th>Synthesis time Original</th>
<th>Synthesis time AET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Height-supervisor</td>
<td>8</td>
<td>71</td>
</tr>
<tr>
<td>Absolute</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>Factorial</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>Resettable counter</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>ePHBRI</td>
<td>8</td>
<td>25</td>
</tr>
<tr>
<td>Producer-consumer</td>
<td>15</td>
<td>4</td>
</tr>
<tr>
<td>Watchdog timer</td>
<td>16</td>
<td>60</td>
</tr>
<tr>
<td>Flight Warning Syst.</td>
<td>17</td>
<td>48</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MRICDF network</th>
<th>Synthesis time Original</th>
<th>Synthesis time AET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Height-supervisor</td>
<td>8</td>
<td>71</td>
</tr>
<tr>
<td>Absolute</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>Factorial</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>Resettable counter</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>ePHBRI</td>
<td>8</td>
<td>25</td>
</tr>
<tr>
<td>Producer-consumer</td>
<td>15</td>
<td>4</td>
</tr>
<tr>
<td>Watchdog timer</td>
<td>16</td>
<td>60</td>
</tr>
<tr>
<td>Flight Warning Syst.</td>
<td>17</td>
<td>48</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MRICDF network</th>
<th>Synthesis time Original</th>
<th>Synthesis time AET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Height-supervisor</td>
<td>8</td>
<td>71</td>
</tr>
<tr>
<td>Absolute</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>Factorial</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>Resettable counter</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>ePHBRI</td>
<td>8</td>
<td>25</td>
</tr>
<tr>
<td>Producer-consumer</td>
<td>15</td>
<td>4</td>
</tr>
<tr>
<td>Watchdog timer</td>
<td>16</td>
<td>60</td>
</tr>
<tr>
<td>Flight Warning Syst.</td>
<td>17</td>
<td>48</td>
</tr>
</tbody>
</table>