Formal Model Driven Software Synthesis for Embedded Systems

Bijoy Antony Jose

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Sandeep K. Shukla, Chair
A. Lynn Abbott
Michael S. Hsiao
Patrick Schaumont
Anil Vullikanti

August 2, 2011
Blacksburg, Virginia, USA

Keywords : Model driven code generation, synchronous systems, software synthesis, polychronous formalism, multi-threading.

Formal Model Driven Software Synthesis for Embedded Systems

Bijoy Antony Jose

(ABSTRACT)

Due to the ever increasing complexity of safety-critical applications, handwritten code is being replaced by automatically generated code derived from a high level specification. Code generation from high level specification requires a model of computation with an underlying formalism and correctness-preserving refinement steps to generate the lower level application code. Such software synthesis techniques are said to be 'correct-by-construction'. Synchronous programming languages such as Esterel, LUSTRE, which are based on a synchronous model of computation are used for sequential code generation. They work on a synchrony assumption (zero time intra-process computation and zero time inter process communication) at the specification level. Early versions of synchronous languages followed an execution pattern where an iteration of software was mapped to an interval between ticks of an external reference clock. Since this external reference tick was unrelated to variables (or signals) within the software, redundant operations such as reading of ports, computation of guards were performed for each tick. In this dissertation, we highlight some of these performance issues and missed optimization opportunities. Also we show how a multi-clock (or polychronous) formalism, where each variable has an independent rate of execution associated with it, can avoid these problems.

An existing polychronous language named SIGNAL, creates a hierarchy of clocks based on the rate of execution of individual variables, to form a root clock which acts a reference tick. We seek to replace the clock analysis with a technique to form a unique order of events without a reference time line. For this purpose, we present a new polychronous formalism termed Multi-rate Instantaneous Channel connected Data Flow (MRICDF). Our new synthesis technique inspects the specification to identify a master trigger at a Boolean equation level to act as the reference tick. Furthermore, we attempt to make polychronous specification based software synthesis more accessible to practicing engineers, by constructing a software tool EmCodeSyn, with a visual environment for specification and a more intuitive analysis technique. Our Boolean approach to sequential synthesis of embedded software has multiple implementations, each of which utilizes existing academic software tools. Optimizations are proposed to minimize synthesis time by simplifying the input to these external tools. Weaknesses in causal loop analysis techniques applied by existing synthesis tools are highlighted and solutions for performing time efficient loop analysis are integrated into EmCodeSyn. We have also determined that a part of the non-synthesizable polychronous specifications can be used to generate correct multi-threaded code. Additionally, we investigate composition of polychronous modules and propose properties that are necessary to guarantee agreement on shared signals.

This project received partial financial support from Air Force Office of Scientific Research, Air Force Research Laboratory and National Science Foundation.

Dedication

To Appachan, Ammachi and Chechi

for their unwavering support

and

to all my friends and well wishers.

- Bijoy Antony Jose

2^{nd} August, 2011, Blacksburg, Virginia, USA.

Acknowledgments

I would like to acknowledge the contributions of various people with whom I was associated with during the course of my Ph.D.

First and foremost, I owe my deepest gratitude to my advisor, Dr. Sandeep K. Shukla. His ideas, guidance, and encouragement was instrumental in the each stage of this endeavor and will remain as a source of inspiration throughout my life. I would like to thank Dr. Patrick Schaumont, Dr. Michael S. Hsiao, Dr. Lynn Abbott and Dr. Anil Vullikanti for serving on my PhD committee. I have had the pleasure of working with several researchers during my Ph.D. and a sincere thank you goes out to Abdoulaye Gamatie, Jean-Pierre Talpin and Jens Brandt. I am thankful for the support received from multiple funding sources such as Air Force Office of Scientific Research, AFRL and NSF for the EmCodeSyn project. I would also like to acknowledge the contributions of Andrew Matusiewicz, Neil Murray and Erik Rosenthal towards development of software tools that were integrated with EmCodeSyn. A special mention of gratitude to my colleagues at Intel Corporation for their help during my internship period.

I have had the opportunity to work with several undergraduate students who have contributed immensely towards the project. I would like to mention Matthew Kracht, Jason Pribble and Lemaire Stewart in particular for their contributions towards this endeavor. Special thanks are due to my fellow lab mates for this enriching academic period - Sumit Ahuja, Bin Xue, Mahesh Nanjundappa, Avinash Lakshminarayana, Gaurav Singh, Deepak Mathaikutty, Syed Suhaib, Hiren Patel, Matthew Anderson, Julien Ouy, Jing Huang, Yi Deng and Hua Lin. My heartfelt thanks to my room mate Roshin Kadannapally for his support, to my fellow Hokies and to batch mates from my graduate and undergraduate college years.

Finally, I would like to thank my family: Dr. Babu T. Jose, Leela P. Jose, Babita Jose, Jimson Mathew and Joemol for their love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Embedded systems have found widespread use in fields such as communication, transportation and medicine. New techniques such as ‘fly-by-wire’ and ‘drive-by-wire’ contribute to the increase in the amount of control software. As applications become more complex, the embedded software that drives these applications also become more complicated. Table 1.1 shows the amount of On-board software in use for different Airbus aircraft models over past three decades [137, 63]. Newer versions of control software try to increase automation, thereby increasing these numbers. Also, platforms which handle control software have changed over the years from simple single issue processing units to multi-core embedded processors (ARM Cortex-A9 [1], Renesas SH-2A DUAL [14]). But due to chances of failure while operating in a distributed environment, safety critical embedded software has lagged behind in utilizing the benefits of parallelism. Therefore the design, development and testing of safety critical software have issues of immense interest.

Software targeting most critical systems have one commonality: they are all reactive, concurrent, and real-time. Writing concurrent software is error-prone and difficult. Due to the safety-criticality of such software, correctness is of paramount importance. As a result, instead of manual programming, formal specification driven code generation is desirable. Formalisms that have rigorous semantic basis, amenable to formal analysis, verification, and refinement are required for such safe code-generation. At the same time for productivity enhancement, engineers need implementation of such formalisms as visual model driven tools that ease the burden of creating and debugging the specification, formally verifying them and finally performing code generation. Figure 1.1 shows the approximate time spent in different stages of an avionics software project without the use of automated code generation. It can be observed that coding and testing phases take up 42% of the total

Parameter	**A310 - 1970s**	**A320 - 1980s**	**A340 - 1990s**
On-board Software (MB)	4	10	20
Errors detected/100KB	500	100	10

Table 1.1: Avionics code information about successive Airbus models[137, 63]

man hours. Model driven code generation aims to reduce time spent in these phases of a software project. 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 Transportation [138]. It recommends the use of sophisticated model driven tools in design, code generation and verification phases to achieve higher quality code with a lower time to market.

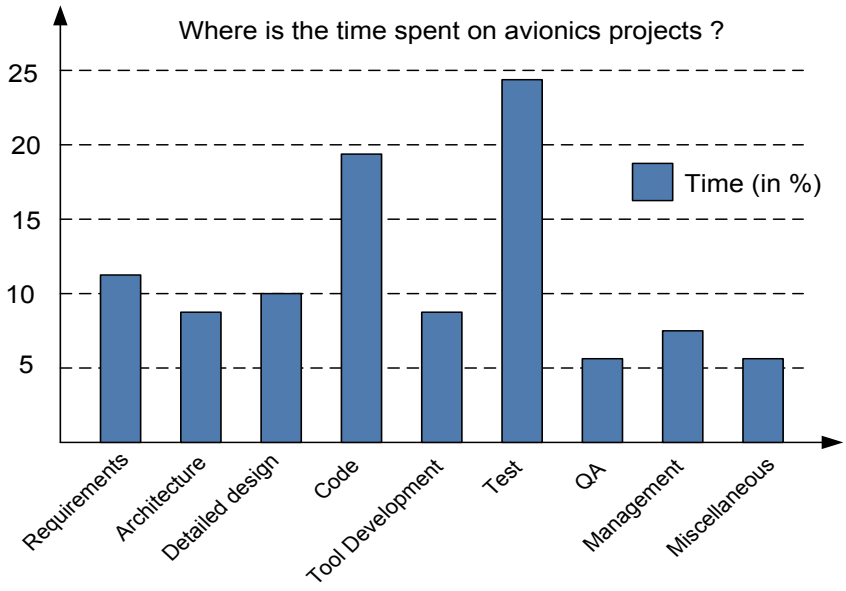


Figure 1.1: Division of time spent in development of avionics software [63]

1.1 Model driven code generation

Model driven code generation tools such as Ptolemy [20], MATRIXx [11] try to provide a starting point for system design in control systems and embedded software. Tools such as LabVIEW [10] and Simulink [8] have been instrumental in driving visual specification methods to the forefront. The methodology in designing software in these tools usually consists of a higher level language/-model (Matlab, Finite State Machine, etc.) which can describe a design without any approximations. Now they are translated into the lower level design (C, RTL, etc.) by the individual tool-specific design flow. The lower the abstraction level, the more control the user can have on the handling of tasks. The limitation of this approach is that identifying parallelism at a lower level is harder and optimization opportunities inherent in the specification stage are missed.

Model driven software tools require a higher level of abstraction which hides the implementation details completely. Figure 1.2 shows a general view on the abstraction levels of the tools, programming languages, and their implementation paradigms. Tools such as LabVIEW [10] and SIMULINK [8] have their own programming languages or formalisms which can be transformed

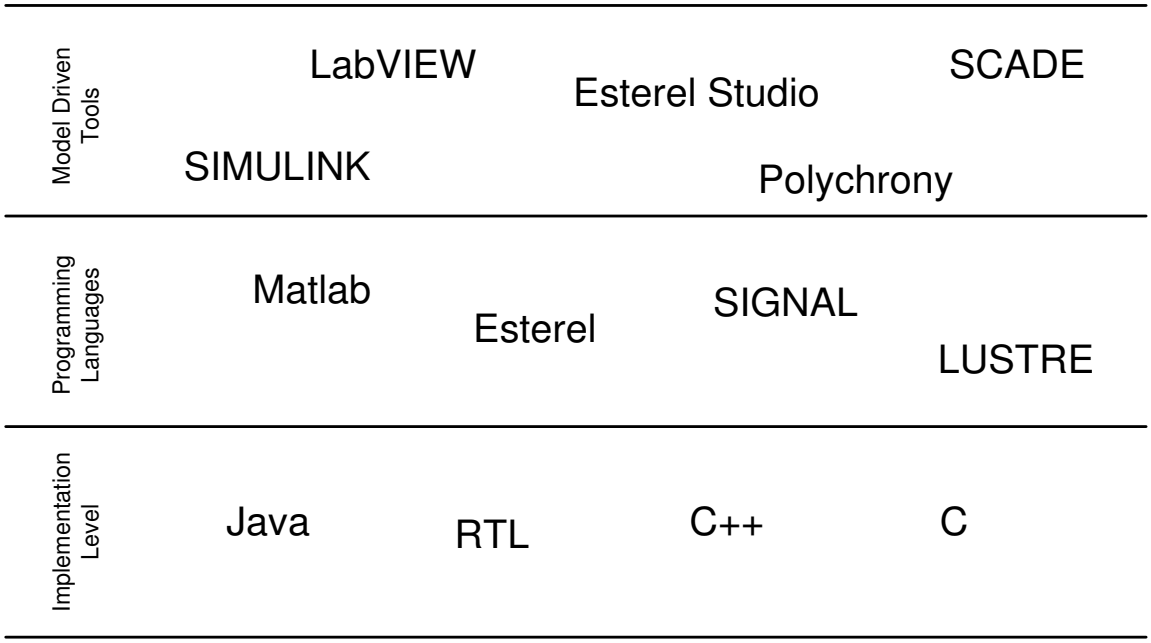


Figure 1.2: Abstraction levels of software languages, directives, utilities and tools

to a lower level code. They often are based on a well known *model of computation* (MoC). Informally, a Model of Computation is the manner in which the computation and communication are being performed. If the design flow methodology is based on sound mathematical basis and correctness preserving refinements are applied, code at the implementation level is said to be *correct-by-construction*. Additional benefit of this methodology is that the verification tasks on the design can also be done at the higher levels of abstraction, which are concise and may require less verification time.

A few of the Models of Computation used to specify embedded software are Petri-nets [116], Kahn Process networks (KPN) [102], Synchronous data flow (SDF) [104] and Finite State Machine with Datapath (FSMD). Model driven tools shown in Figure 1.2 such as Simulink/Stateflow are based on Statecharts [77], which is a variation of hierarchical state transition diagrams. Synchronous programming languages such as Esterel [43], Lustre [74], SIGNAL [71] form the basis of tools such as Esterel Studio [131], SCADE [62] and Polychrony [61] respectively. Properties of synchronous languages such as synchrony, reactive response, concurrency and deterministic execution cater to the needs of a designer of safety critical embedded software and hence tools based on synchronous MoC deserve a closer look.

Synchronous Programming Languages

The synchronous programming model represents an MoC that can be used to specify reactive embedded software and to synthesize sequential implementations. The essence of synchronous MoC is the *synchrony hypothesis* which abstracts the computation and communication time. The computation and communication in a synchronous specification are assumed to be instantaneous, while

the order of arrival of events and execution of computation is preserved. Synchronous specification has the concept of an external global clock, whose tick triggers the computation of a new set of outputs and before the following tick occurs, the computation and communication of these outputs are completed. This global clock has no relation to the hardware clock and it acts as a reference for each round of input-output events. Languages such as Lustre [74] and Esterel [43] follow the synchronous MoC. Within the synchronous MoC, there is an alternate approach adopted by the language SIGNAL [71] where the presence of the external global clock is not assumed and a reference clock is formed from the specification. Here each variable is associated with its own clock and a root clock is found which has a synchronous tick for every tick occurring in the system. This approach where each variable has its own rate of computation is called multi-rate or *polychronous* formalism.

Synchronous formalism based tools can generate deterministic software code for safety critical applications. Strengths such as concurrency, reactive response and deterministic output from synchronous specification makes synchronous software code synthesis an attractive proposition for safety critical embedded applications. Synchronous specification is concurrent, albeit has compilers that generate sequential code as output. Since variables in synchronous specification may or may not be computed between two clock ticks based on their guard conditions, visualizing flow of code becomes a hard task. Hence model driven visual software tools are an important value-add to synchronous languages.

Applying synchronous programming or for that matter any new MoC as the basis for a software synthesis tool will require an alternate way of conceptualizing a design by the programmer and learning of new languages for specifying a design. Adopting model driven code generation techniques will also incur costs of shifting to new programming methodologies. But the shift to model driven code generation is justified by metrics that show that the returns are far greater than the costs incurred. It is reported that the use of SCADE, a model driven code generation tool from Esterel Technologies [62] has been successfully used for synthesizing 70% of the code for a Flight Control Secondary Computer unit of Airbus 340/600 [137]. The use of SCADE resulted in a 50% reduction in development cost [63]. Table 1.1 shows the errors detected per 100Kbytes of code for successive Airbus projects in their testing phase.

Research into model driven tools, synchronous languages and code generation poses challenges in different topics which decide the implementability of synchronous systems. Our research is oriented towards the specification, analysis and synthesis of embedded software. We divide these research topics into four areas of interest:

1. Composition of synchronous systems
2. Sequential software synthesis techniques from polychronous formalism
3. Multi-threaded software synthesis from polychronous specifications
4. Causal loop analysis on polychronous specifications

Rest of this chapter will introduce these particular areas of interest, discuss the problems faced in these areas and list our contributions in each of them.

1.2 Composition of Synchronous Systems

Safety-critical embedded applications are often distributed. Synchronous programming models make the synchrony assumption (zero time intra-module computation and zero time inter-module communication), so that the models are easier to verify. Once verified, models built with such assumptions need to be distributed over an asynchronous communication based platform which brings out the challenge of Globally Asynchronous and Locally Synchronous (GALS) design. The correctness preserving refinement of a fully synchronous model onto a globally asynchronous communication media implies that various restrictions be imposed on the synchronous model. In the realm of synchronous programming model, a property called *endo-isochrony* was proposed in the 1990s [38, 39]. If satisfied by a collection of synchronous modules or intellectual property (IP), it would guarantee that the distributed deployment of the modules across asynchronous media would be safe. By being 'safe', we mean that the data flow behavior of the collection of modules when working under synchrony assumption, and when working in asynchronously communicating scenario, would be the same.

The *endo-isochrony* condition can be assessed separately as *endochrony* and *isochrony* conditions which are informally introduced here. 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. In endochrony, sequentiality has been a prime precondition, leading to a restrictive class of synchronous models which satisfy this condition. In later works [119], this condition was relaxed so that strict sequential implementability of the modules were not primary. Local nondeterminism that does not disturb the global determinism of the module I/O were allowed, leading to the notion of *weak-endochrony*. The class of weak-endochronous modules strictly subsumes the class of endochronous modules. *Isochrony* is a condition that requires communicating synchronous components to agree on the presence or absence of shared signals at each time instant [129]. Each component has a clock dependency for all its signals and the rate of arrival of events on shared signals should be in accordance with the expected arrival at that shared signal. If the isochrony property is violated, then the communication between the processes will need specialized protocols such as hand-shake or other synchronization. A system whose component's computation is endochronous and communication is isochronous are called as endo-isochronous systems.

Contributions towards Composition of Synchronous Systems

The research efforts in the field of synchronous design and composition have evolved over the years to form properties such as endochrony, weak-endochrony and isochrony. Lacking in these efforts has been a comprehensive study which analyzes IPs and their composition with respect to synchronous programming languages. This would require analysis of mathematical definitions or

assertions in the context of a particular synchronous language and formulation of tests for each property. In the absence of these tests for properties, they are of no practical use. Programming tools based on synchronous languages such as SIGNAL require programs to be endochronous, and thus do not allow a class of weak-endochronous programs that can give a deterministic output. Remedying this situation is important, since weak-endochrony being a relaxation of endochrony property, can allow a larger set of designs to be implemented.

In our work, we generalize the sufficiency conditions for isochrony [38, 39] to give a characterization in terms of clock relations between signals. In another direction, we provide a sufficiency condition for weakly endochronous modules which require a modification of the definition used for endochronous components. We find conditions on clock relations either directly coded as clock constraints, or indirectly inferred from the data flow equations, which would imply that the GALS implementation of the original synchronous model under analysis is correct. The contributions towards composition of synchronous systems can be summarized as:

1. Formalize the conditions for isochronous composition and propose a *communication structure* involving the parameters required for composition. The assertions we propose based on the communication structure will serve as a sufficient test for the composition of IPs [100].

2. We show that weak endochrony relaxes the strict conditions for composition of IPs and correctness can be guaranteed for communication towards a weakly endochronous process. We also define *directional isochrony* based on signal relations, which provides us with a special set of rules for correctness of communication in a single direction [100].

1.3 Sequential Software synthesis techniques from polychronous formalism

Synchronous programming languages such as Esterel[43], SIGNAL[71], LUSTRE[74] etc., have been successful in specifying embedded reactive software and in generating sequential software code from them. Even though broadly classified as synchronous languages due to the underlying synchrony assumption, these languages often have very different underlying formalism [40]. For example, Esterel, and LUSTRE programs have the notion of totally ordered global tick (or global clock), which provides a sequence of global synchronization points for the reading of inputs and the corresponding reaction to start. During the execution of the software synthesized from those languages, one has to provide some external means to indicate these ticks or synchronization points. On the other hand, SIGNAL is a data flow synchronous programming language which does not assume the existence of any external trigger for reacting to the inputs. It is paced by the rate at which data arrives or that of some other internal events occur. The freedom from the notion of a global synchronization or ticks leads to a *multi-rate* system specification and hence SIGNAL is known to have a *polychronous* formalism [61]. Lately, both Esterel and LUSTRE have introduced multi-rate versions for their language which allow signals to be associated with their own clocks.

Nevertheless, both multi-rate versions do operate with an external global software clock, with each signal acting on its computation in relation to this global clock. A few examples are presented below to explain why we favor multi-rate specifications over single rate or globally synchronized model of computation.

Motivating cases for polychronous formalism

Embedded software systems interact with their environments by sampling inputs (continuous inputs such as temperature, pressure etc.) or by getting discrete inputs from other digital systems, infinitely. Hence all the inputs, internal variables and outputs of such systems can be thought of as infinite streams of values. If one has a global discrete clock, one could also associate time stamps to each occurrence of values on each stream. Here, we do not assume any such a priori global clock, and hence occurrence of a new value on a variable is termed an *event*. Depending on the data type of the variable, events have *values*. Following the existing literature, we term each variable in such embedded systems a *signal*. More formally, a signal s is an infinite stream of events, each event associated with some value $v_1, v_2, ...$ etc. .

Example 1.3.1 (Priority-Merge)**.** *Let* `a` *and* `b` *be two signals, and we need a software that can merge these two streams of events into one stream such that if* `a` *and* `b` *have events at the same instant, the value from* `a`*'s event will be sent to the output. If* `b` *has event at an instant when* `a` *does not have an event, then only the value from* `b`*'s event will be sent to the output.*

In a single clocked specification language such as Esterel, time is globally divided into discrete instants at which inputs are sampled, and outputs are computed with no time delay. So the term 'instant' in Example 1.3.1 refers to discrete time point in a linear global time line, at which sampling `a` and `b` determines if any events are present and if so what their values are. Thus if `a` is absent at the sampling instant, and `b` is present, then output at that instant will take the value from the stream `b`. However, in a multi-clocked specification, no externally defined global linear time line is present, and hence there is no unique implementation for Example 1.3.1. One example would be that the software waits for events on both inputs, and if an event occurs on `a`, it outputs that event. However, if an event occurs on input `b`, it outputs event on `b`. But an 'instant' in the specification is a period in real execution. So, if `a` has an event after after `b` has an event within the same period, one should output event on `a`. Since the length of this period is not predetermined, the implementation will be nondeterministic. Does that mean multi-clock specifications are problematic? We will see in later chapters, that is not the case. Now consider another example which is a slightly modified version of Example 1.3.1.

Example 1.3.2 (Mutually Exclusive Signals)**.** *Suppose, events on* `a` *or* `b` *are always accompanied by an event on* `c`*. It is known that every time an event on a signal* `c` *arrives with value* `true`*, an event on* `a` *arrives. Also, when an event on* `c` *arrives with a value of* `false`*, event on* `b` *arrives.*

Here the implementation should remain suspended until an event occurs on `c`. Later determine based on the value of `c`, which of the two streams (`a` or `b`) must be sampled. If no event on `c` occurs, the system does not react at all. Now we see the advantage of multi-rate specification.

In a global synchronization based model of computation, every the time global tick occurs, at least `c` must be sensed. Hence the program needs to wake up repeatedly even when no event on `c` occurs, leading to less efficient implementation (e.g, the system cannot go to sleep mode). A multi-rate specification and its true implementation need to wake up only on events of `c` and perform computations.

The above examples are to illustrate cases where global synchronization based specifications unnecessarily restrict some obvious optimization opportunities that can be exploited using a polychronous formalism. The relative popularity of Esterel and LUSTRE is due to the simplicity of its globally synchronized execution model. Multi-rate models are not restricted due to their independence from an external global clock, in fact they give more freedom for compilers to choose from different schedules for computation. These refinement choices for the designer makes polychronous specification more amenable to optimized code synthesis. Since complex semantic exposition of the SIGNAL language restricted its widespread use in the industry, our attempt is to make polychronous specification based software synthesis more accessible to engineers, by proposing an alternative model with different semantic exposition and analysis techniques.

Contributions towards sequential software synthesis techniques from polychronous formalism

We provide an alternative for multi-rate specification formalism, with analysis and synthesis techniques that are very simple to understand for designers. Our prime implicate based implementation from polychronous specification alleviates the bane of complicated semantic theory of SIGNAL, based on clock equations, without compromising the rigor. Specifications must be as abstract as possible, leaving more choice to tools for optimization. This is the philosophy with which we have developed MRICDF as a user friendly polychronous formalism, a theory of implementability, and a visual environment to support such modeling and synthesis.

1. An alternative specification language and formalism for multi-rate reactive deterministic embedded software systems which we call MRICDF or *Multi-Rate Instantaneous Channel-Connected Data Flow* actor network model [93, 94].

2. Sequential implementability of a polychronous model is analyzed and a static analysis procedure is proposed (*Epoch Analysis*) to realize MRICDF networks into embedded software. A method for formulating implicit or derived triggering conditions for polychronous models is proposed. Proofs are provided for the existence of a master trigger for deterministic software implementation from a polychronous model. A prime implicate technique for identifying the master trigger and a unique order of instants in an MRICDF network is proposed [94, 95].

3. Constructed a code synthesis framework *EmCodeSyn*, for enabling MRICDF driven programming for synchronous system design. EmCodeSyn will provide a visual design and debug facility for MRICDF based synchronous software synthesis [88, 4].

4. Optimization techniques for minimizing the software synthesis time due to prime implicate computation is explored. Restrictions are imposed to ensure that the sequential implementability criteria of MRICDF networks is not altered by these optimizations. A simpler

set of inputs to prime implicate generator was obtained by our Actor Elimination Technique which minimized prime implicate computation time and consequently overall synthesis time [86, 95].

1.4 Multi-threaded software synthesis from Polychronous Specifications

In order to exploit the emerging multi-core processors, creating multi-threaded applications is a necessity. However, resolving concurrency, synchronization, and coordination issues, and tackling the non-determinism germane in multi-threaded software is extremely difficult. Ensuring deterministic behavior and correctness with respect to the specification is necessary for safe execution of such a code. It is desirable to synthesize multi-threaded code from formal specifications using a provably 'correct-by-construction' approach. Given a polychronous specification of a reactive embedded computation, under what condition one can generate deterministic multi-threaded C-code whose input/output behavior exactly matches that of the specification? Also, we consider two additional constraints for this synthesis problem: (a) The granularity of threading can be arbitrary with implications to the performance of the multi-threaded code, but the behavioral equivalence should be maintained. (b) The threading must be preemptive and hence no assumptions can be made on the order of thread executions.

Contributions towards Multi-threaded software synthesis

Existing compilers for polychronous languages have implemented deterministic multi-threaded code generation. Here determinism is present at the final output of the IP, while the internal behavior of the IP could change according to the scheduling of the multi-threaded code. We investigate the existing threading structures of the SIGNAL compiler and demonstrate how a tool for sequential code generation can be used to generate multi-threaded code with different thread granularity. Endochrony remains to be a necessary condition for multi-threaded code generation for the Polychrony/SIGNAL complier. We analyze under what condition a polychronous specification can be synthesized into multi-threaded C-code, while preserving its semantics. In particular, we investigate cases where weak-endochrony programs can be used to generate deterministic C code.

1. We show why endochrony is not a necessary condition for deterministic multi-threaded C-code generations from SIGNAL specifications. We also show cases where weak endochrony is sufficient to generate multi-threaded code. This also implies a compositional synthesis methodology, since weak-endochrony is compositional [97].

2. We show through examples how the Synchronous Flow Dependence Graph (SFDG) data structural representation of SIGNAL specifications can be exploited to infer the threading structures best suited for a particular specification's multi-threaded implementation [97].

3. In contrast to a micro-threading scheme used in Polychrony compiler and SFDG strategy, we

explore the possibility of a process based coarse-grain multi-threading strategy for SIGNAL. We use the sequential code generation tool of the Polychrony compiler in a non-invasive manner to demonstrate this alternate multi-threading technique, which gives the control of threading granularity to the programmer [85].

1.5 Causality analysis on polychronous specifications

As opposed to single clocked synchronous programming paradigms, polychronous formalism allows specification of concurrent data flow computation on signals such that various data flows can evolve asynchronous with respect to each other. Data flow computation within an instant happens based on the implied data flow order. This order or data dependency often varies from one instant to another. Thus determining if there is an instant at which the data flow order forms a causal cycle is an important problem. In the current polychronous compilers, this is solved without due effort, by rejecting any program which has a buffer-free structural cycle. However, a clocked dependency graph can be used to construct logical constraints representing the instants with a possible causal loop. The satisfiability of such constraints would imply that such a loop is realizable and hence the specification has a possible deadlock. The reachability of this instant with a given set of initial conditions would verify if the program should be rejected.

Contributions towards causality analysis techniques

In the past, the work on such constraints and their satisfiability has not been implemented even though for pure Boolean signals and clocks this could have been done using a satisfiability solver. With the advent to SAT modulo theory (SMT) solvers, this can now be extended to a more general class of specifications. Moreover, model checking on an abstraction of the specification can provide more information about the reachability of instants at which cyclic data dependency is realized. Moreover, model checking on an abstraction of the specification can provide more information about the reachability of instants at which cyclic data dependency is realized. Our proposed causality analysis technique presents an improved polychronous synthesis tool accepting a much larger class of specifications than could be done before. In our experimental results, we demonstrate the capabilities of our causality analysis methods and show that our synthesis tool performs better than previous strategies.

We also look into software synthesis opportunities for verification tools such as the SMT solver. The prime implicate technique of implementing Epoch Analysis is one methodology that can be incorporated in our visual framework. We investigate the possibility of identifying the master trigger for MRICDF networks using SMT solvers to speed up synthesis.

1. We illustrate the weaknesses of current causality analysis techniques and motivate the need to move onto more expressive and powerful structures which expand the set of implementable polychronous programs using a software tool. Data domain constraints and non-realizability of a clock constraint at which a data dependence loop is formed cannot be properly utilized in

the absence of expressive theory of data domains, arithmetic, and propositional satisfiability. A new SMT based approach to causality analysis is proposed which can accommodate clock and data dependencies from a polychronous specification [84].

2. In existing tools, the absence of these techniques leads to overly pessimistic compilation, rejecting many polychronous specifications with non-realizable causal loops. We distinguish different types of causal loops in polychronous specification and a heuristic tailored for faster detection is implemented which uses specific verification techniques for each loop type [83].

3. Prime implicate based technique implements Boolean theory approach to sequential synthesis by computing all possible prime implicates of a polychronous specification and later identifying master trigger signals among them. Recent enhancements in SAT/SMT solvers is exploited by posing identification of master trigger as a satisfiability problem. Synthesis time while using an SMT solver for master trigger identification is found to be significantly better than direct prime implicate computation [83].

1.6 Dissertation organization

This dissertation is organized according to the three areas we address as shown in Figure 1.3.

Chapter 2 discusses the background information on Models of Computation, synchronous programming, polychronous formalism, SIGNAL language etc.

Chapter 3 presents the related work done regarding synchronous languages, polychronous formalism and code synthesis tools based on different models of computation.

Chapter 4 analyzes the composition of synchronous systems. We define the mathematical preliminaries required for communicating structures of synchronous IPs and check for the correctness of composition of synchronous IP blocks using them. Further, we introduce the notion of directional isochrony which provides sufficient conditions for safe communication between modules in a single direction.

Chapter 5 analyzes under what condition a polychronous specification can be synthesized into multi-threaded C-code while preserving its semantics. We also discuss how the synchronous data flow graph structure for a polychronous specification can be used to infer the threading structure of the resulting C-code.

Chapter 6 presents an alternate polychronous formalism termed Multi-rate Instantaneous Channel connected Data Flow (MRICDF) actor network. We introduce Epoch Analysis as a technique to synthesize correct sequential embedded software from MRICDF models. We propose a prime implicate based technique for implementing Epoch Analysis for MRICDF networks.

Chapter 7 introduces EmCodeSyn, the visual specification and synthesis tool for MRICDF models. The design flow methodology and case studies for design and debug of MRICDF models is

illustrated in this chapter.

Chapter 8 will address the optimizations for speeding up synthesis using EmCodeSyn. The properties of an MRICDF network that need to be preserved while simplifying its Boolean equations for faster prime implicate computation are discussed. Actor elimination rules are drawn up for removal of actors that do not affect prime implicate results. Proofs are provided to show why these elimination rules do not affect synthesis results.

Chapter 9 discusses the opportunities for applying verification tools such as SAT/SMT solvers in polychronous software synthesis. Elements of sequential synthesis conditions of a polychronous specification such as causal loop detection is shown to be improved with the application of the right verification tool for each casual loop type. An alternative implementation of Epoch Analysis where implementability conditions are posed as satisfiability problems is also discussed.

Chapter 10 concludes the dissertation with thoughts on possible extensions to this work.

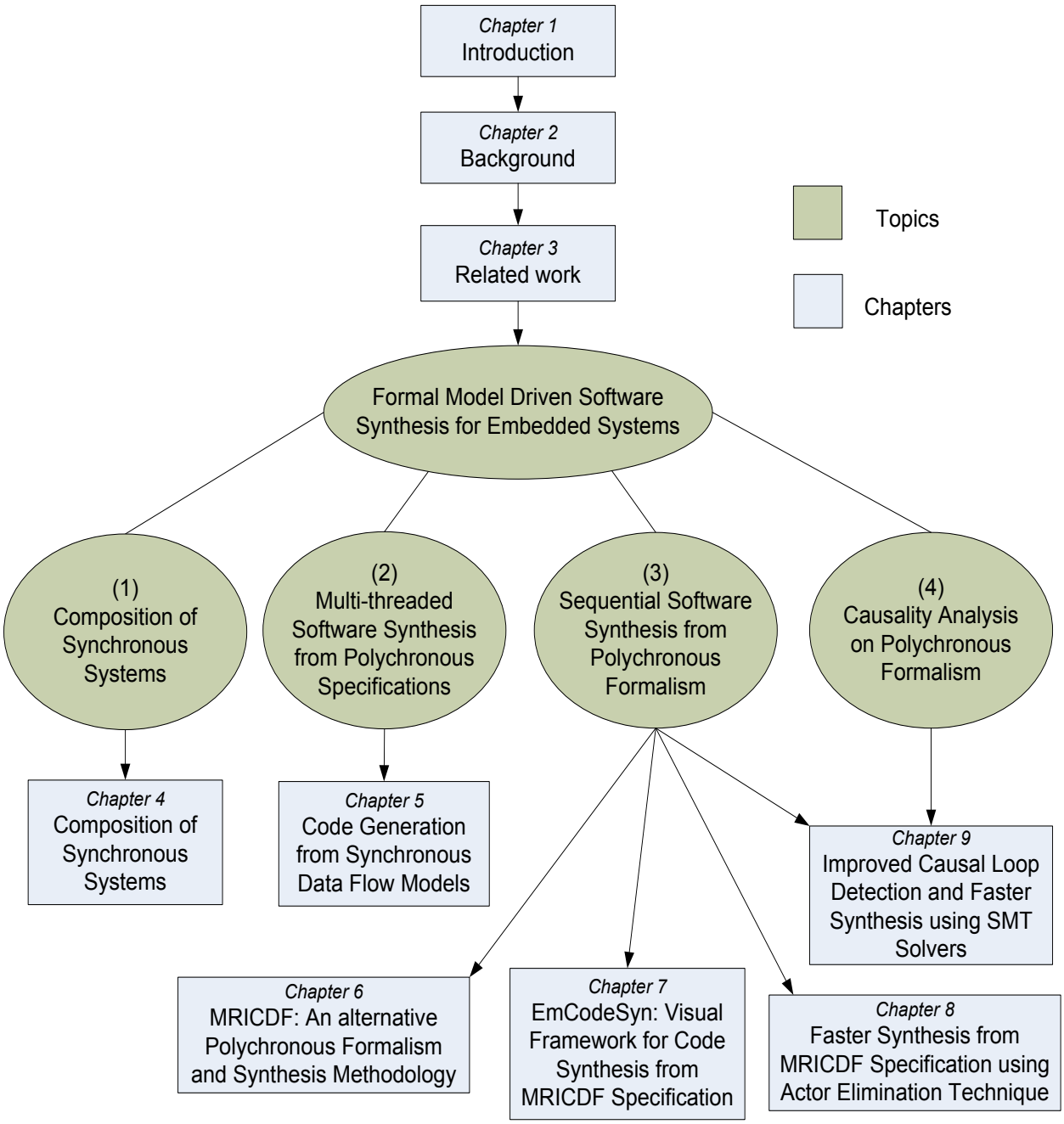


Figure 1.3: Thesis organization

Chapter 2

Background

In the last few years, parallel computing has claimed its niche in improving performance and power trade-off [80] for general purpose computers. So, it is no surprise that multi-core architectures will make inroads into the embedded processor markets as well. Recent works have shown that parallel operation and intermittent context switches have a significant impact on the power numbers [92]. As background information, this chapter introduces the reader to the different programming models used for multi-threading and the different models of computation used for code synthesis. We also discuss various synchronous programming languages used for code synthesis and in particular, the polychronous language SIGNAL.

2.1 Programming models for concurrent code generation

Currently, major processor vendors have already released products containing multiple cores on a single die [79], however, one must employ concurrent programming models when designing their programs to exploit the architecture with multiple cores in such products. One such programming model, and one that most designers are familiar with, is the multi-threaded programming using thread libraries. But, complex tasks require planning and division of tasks at a higher abstraction level and refinement into lower levels of abstraction. Hence conceiving a task using models of computation which can fully express concurrency is very important. Here we first introduce some of the thread libraries and then discuss a few programming models for concurrent code generation.

2.1.1 Multi-threaded programming

Thread libraries are one of earliest APIs available to perform multi-tasking at Operating System level. POSIX threads [5] and Windows threads [9] are some of the APIs used for multi-threading in Unix and Windows OS. With the help of the specialized functions defined in these libraries,

threads (or flows of control) can be generated which can execute concurrently. Thread libraries work at a lower level with tight control over the thread execution, which gives the programmer more control, but with greater responsibilities.

The implementation of POSIX threads and Windows threads are different, but their overall programming model is the same. A single flow of control or main thread is forked out into separate flows of execution. The *fork and join* threading structure for libraries such as POSIX threads or Windows threads is shown in Figure 2.1. The main thread shown in Figure 2.1 has been separated into five flows, each thread with a unique ThreadID. These threads have associated function calls which specify the operations they will perform and attributes which is the data passed onto the function. The main thread (Thread A) can be used for computation as well, but in common practice the intention is to control the fork and join process. The join is used to wait for the completion of execution in different flows. The functions executed will return the data and a single flow of control is resumed.

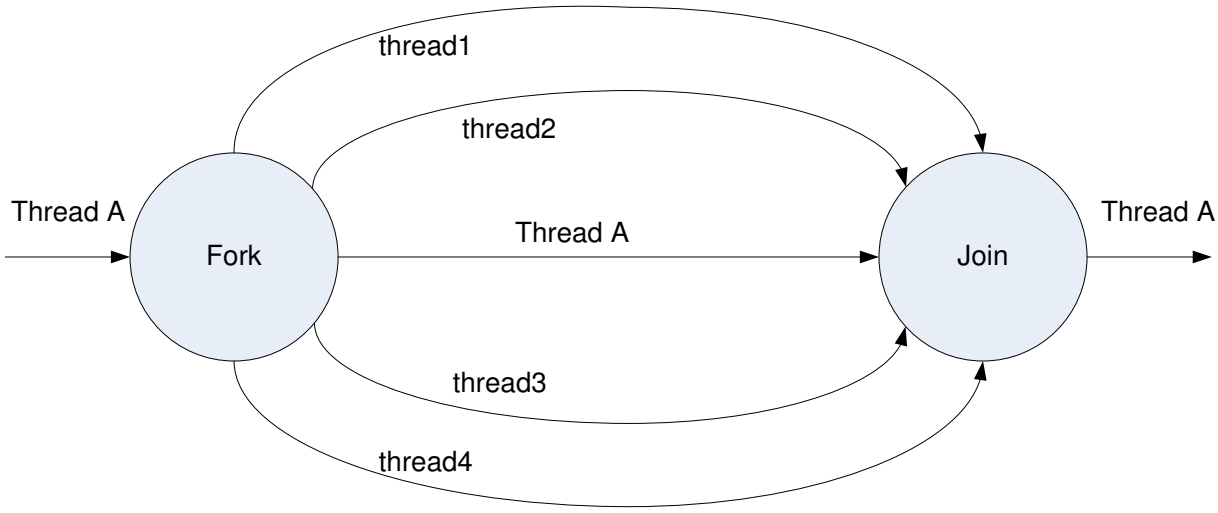


Figure 2.1: Threading Structure of fork-join model

Tasks in an Operating System (OS) are modeled as processes which follow a threading model. They are scheduled by the kernel based on criteria like priority, data integrity, etc. OS executes these kernels leaving the user with less control over the execution. The hardware thread runs on each core or a virtual core is called a *kernel thread* and the code provided by the user is called a *user thread*. The parallel execution of the threads can be in One to One, Many to One, or Many to Many ratio between user and kernel threads. In the absence of multi-core processors, the threading has to be performed by time sharing of a hardware kernel thread between the software user threads. This can still outperform the single threaded execution model, because a thread that is not running can still be performing a memory operation in parallel using peripherals of the processor. A *work distribution* model from user threads to kernel threads for multi-core or

multi-processor systems is shown in Figure 2.2. In the figure, n threads are distributed among m execution cores by the scheduler. Here the focus is on maximizing the utilization of the processing cores by an efficient scheduling algorithm. The cores (homogeneous or heterogeneous) are not allowed to remain idle by the scheduler. Another programming model using threading libraries is the *pipeline* model. The work done by each stage in the pipeline is modeled as a thread and the data being acted upon changes with time. Every pipeline stage needs to be executed simultaneously for optimal performance results. Figure 5.5 shows a three stage pipeline, each stage having its own execution thread. Since this model involves transfer of data from one stage to the other, additional synchronization constraints need to be considered. The difference between the two programming models is in the flow of data. Pipeline model has separate data and instructions with the data moving across stages performing repeated operations. Work distribution model has data tied to the complete set of operations to be done on them which is assigned to one or many cores.

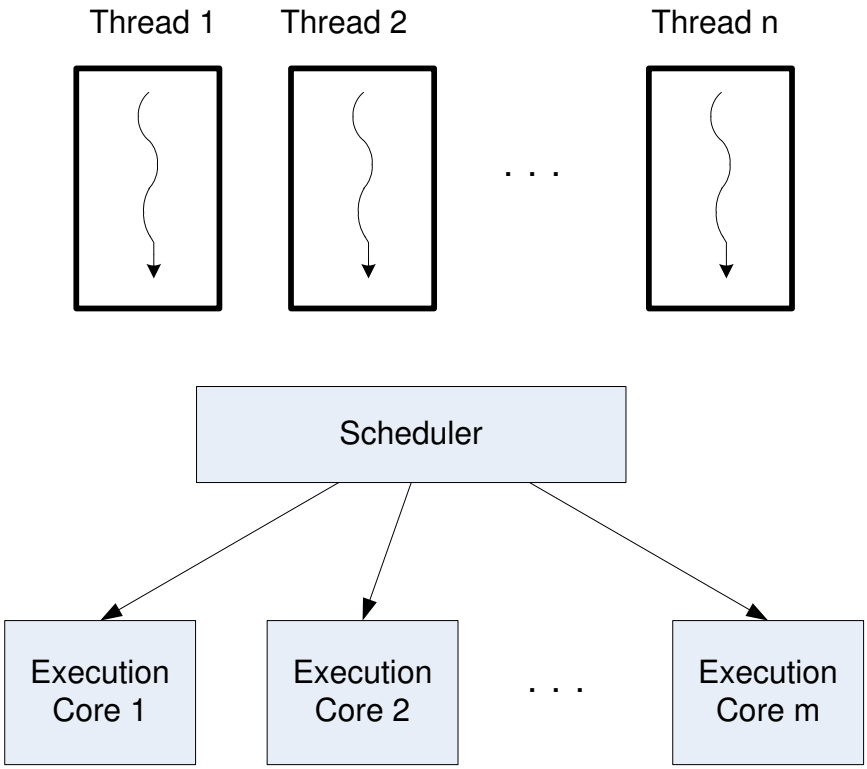


Figure 2.2: Work Distribution model

POSIX threads (Portable Operating System Interface for Unix) or Pthreads in short are APIs for operating systems like Linux, MacOS etc. It consists of header files and libraries which has pthread functions to create, join and wait for threads. Each thread will have its own *threadID*, which is useful for the user to allocate functions and data for their tasks. A POSIX based threaded code for upcount and downcount of a protected variable in no fixed order is shown in Listing 2.1. The master thread is the `main` function used to fork and join threads and is devoid of any functional computation. $pthread\_create$ is used to create threads which call functions `countUp` and `countDown`.

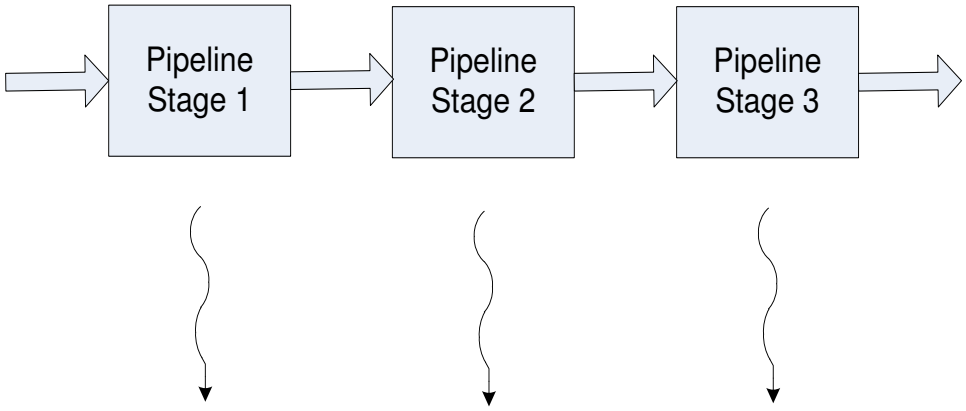


Figure 2.3: Pipeline threading Model

There are no attributes to be sent to the functions, hence only the function names are associated with the threadIDs i.e. $thread1$, $thread2$. The `countUp` function increments the variable a and `countDown` function decrements it. The protection for the shared variable a is provided by POSIX primitives which will be discussed in a later section. Please note that this example is a simplified form to show the threading functions. The thread create and join operations are usually accompanied with check error statements to abort operation in the case of an error. Windows threads are set of APIs provided by Microsoft Corporation for their Windows operating system. This facilities provided by this API is more or less similar barring a few points. In Windows threading APIs, objects are accessed by their *handle* and the object type is masked. Object types can be threads, synchronization primitives, etc. One can wait for multiple objects of different type using the same statement and thus remove the additional *join* statements in the Pthreads case. But some would consider this a disadvantage as the code is more ambiguous when used with *handle*.

Listing 2.1: Pthread code for fork-join model

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4
5  int a = 0;
6  pthread_mutex_t myMutex = PTHREAD_MUTEX_INITIALIZER;
7  void countUp(void *ptr);
8  void countDown(void *ptr);
9
10 int main()
11 {
12   pthread_t thread1, thread2;
13
14   pthread_create( &thread1, (pthread_attr_t *) NULL,(void *) countUp, (void *) NULL);
15   pthread_create( &thread2, (pthread_attr_t *) NULL,(void *) countDown, (void *) NULL);
16
17   pthread_join(thread1, (void *) NULL);
18   pthread_join(thread2, (void *) NULL);
19   return 0;
20 }
21
22 void countUp(void *ptr)
```

```
23  {
24    for (int i=0, i<5, i++)
25    {       pthread_mutex_lock(&myMutex);
26      a = a+1;
27      printf("Thread1: %d\n", a);
28      pthread_mutex_unlock(&myMutex);
29    }
30  }
31
32
33  void countDown(void *ptr)
34  {
35    for (int i=0, i<5, i++)
36    { pthread_mutex_lock(&myMutex);
37      if(a > 0)
38      {
39        a = a-1;
40        printf("Thread2: %d\n", a);
41      }
42      pthread_mutex_unlock(&myMutex);
43    }
44  }
```

The arguments about choosing APIs may not be conclusive but there are common issues that needs to be paid attention to while working on this level of abstraction. The highest importance is for the mutual exclusion property required while accessing shared data. There are sections in threads which needs sequential update operation on data to maintain data integrity. In the Listing 2.1, the variable a needs to be provided with sufficient protection to avoid conflict between read and write operations of the two functions. Thread APIs have several kinds of objects like mutex, semaphore, critical section etc. which provide mutual exclusion property. These objects ensure that there is a lock placed on the critical piece of data and the key is given to only one thread at a time. A detailed discussion of the data access issues in the threading or transaction based execution is given in the next section.

2.1.2 Protections for Data Integrity in a multi-threaded environment

In multi-threaded software, whenever there is sharing of data, out of order update operations on shared memory is a concern. When multiple threads are allowed to write on a memory location, the writes performed on it should be in-order. Even when the order of access is fixed, completion of a write operations need to be ensured. There could be read-write conflicts as well, if multiple read operations are performed on a critical section while a write operation is ongoing. The read value from the memory location is now ambiguous, so ordering of write operations is not enough to have correct operation. If two threads (Thread A and Thread B) are allowed to enter a critical section, the final value of the shared memory location is unpredictable. If multiple threads are allowed to compete for access to a data point or a memory location, we have a *race condition* delivering unpredictable results. For deterministic result for each run of the code, the critical sections have to be protected by mutual exclusion primitives.

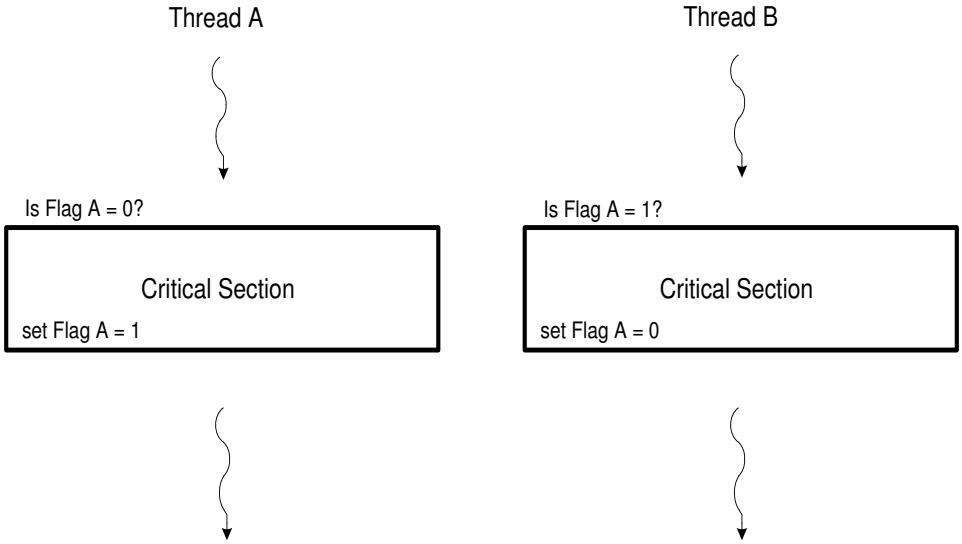


Figure 2.4: Scheduling threading Structure

Mutual exclusion primitives for deterministic output

The solutions for avoiding corrupted data are based on mutual exclusion property. This strategy is based on giving a single thread access to each critical piece of data. The implementation of such a protocol can be based on a flag based entry and exit of the critical section. Figure 2.4 illustrates the access of two critical sections of code in separate threads, which also shares the same variables. If an entry and exit flag is added to the beginning and end of the critical section of each thread as shown in Figure 2.4, we can have synchronized update of the shared data. The flags are used as a constraint at the entry to the critical section to verify if any other thread in the critical section at that point. A failure of this model is in the protection of the flags that have to be shared amongst the threads. An entry to the critical section does not ensure that the flag set/reset operation has been done in sequential order, and hence the integrity of the flags is questionable. Also a read on the flag should not be processed while a write has been issued on the same flag by another thread. Such protection can be provided only by using *atomic* operations on registers, which sequentialize the write/read operations according to their order of assignment.

In POSIX standard, a *mutex* object can be utilized to perform the atomic operations. A lock and key mechanism is implemented around the protected mutex variable (say z). Each thread will try to obtain the key to access the locked variable. Functions can be called from threads to lock until allowed access ($pthread\_mutex\_lock(z)$), or to try lock and return if not allowed access ($pthread\_mutex\_trylock(z)$). An unlock operation ($pthread\_mutex\_unlock(z)$) is performed after the critical section operations are performed. Another synchronization object is *semaphore* (counting mutex) developed by E. W. Dijkstra [58]. A fixed set of threads can enter the critical section and the number of accesses is maintained by the upcount (entry) and downcount (exit) of

the semaphore. Here one semaphore is regulating accesses to multiple resources. Along the lines of POSIX synchronization objects, a *critical section* object can be used in Windows threads. There exists other synchronization primitives like *monitor* from Hoare [78] or *event* which are suitable for specific locking and notify situations. They check for new events on a protected variable and notify a set of threads waiting for that particular operation to be completed, to resume their individual work. Primitives defined by POSIX/Win32 standards might ensure that the critical section is devoid of any race conditions. But deadlocks (multiple threads waiting for access) or livelocks (multiple threads starved of resources) can still appear during the execution of multi-threaded code if the primitives are not carefully used by the programmer. Algorithms based on these mutual exclusion primitives have been proposed like Peterson's algorithm, Lamport's algorithm, Dekker's algorithm, Bakery algorithm etc. summarized in [135], which will assure programming models are free of race conditions, deadlocks, livelocks etc.

2.1.3 Models of Computation for code generation

In this section, we outline a few relevant related works on code generation from concurrent specification of computation. Specifically, we discuss how concurrency is expressed by some of the relevant Models of Computation (MoC).

Kahn process networks

Kahn process networks (KPN) [102] is a primary specification formalism for dataflow between processes. KPN model has been proposed to capture concurrent processes in a system. In a KPN, the processes (nodes) communicate by unbounded unidirectional FIFO channels (arcs), with the property of non-blocking writes and blocking reads on channels. A blocking read means is one which will wait for the data to arrive before doing any other operation. A non-blocking write is one which will not wait to see if output storage is empty before the write operation. Due to these properties a Kahn process cannot wait for multiple inputs channels at once. Also if the order of inputs arriving at a process is preserved, their timing is immaterial, since the sequence of outputs will remain the same. These ideal conditions can be consistent only with the use of unbounded buffers between Kahn processes.

A sample KPN model is shown in Figure 2.5 where 6 processes ($P_1, P_2, .., P_6$) and their intermediate FIFOs are shown. In an ideal environment with infinite size FIFOs, each Kahn process is a concurrent task with asynchronous communication. Here, the processes P_2 and P_5 (also P_3 and P_6) are parallel to each other, since they do not have a dependency between them. With these concepts in mind, we can implement the given KPN model in multi-threaded programming model as follows:

$$[P_1 \; ; \; (P_2 \; || \; P_5) \; ; \; (P_3 \; || \; P_6) \; ; \; P_4]^{*1}$$

[1]Here ';' represents the sequential composition, '||' represents the parallel composition and '$*$' represents repetition

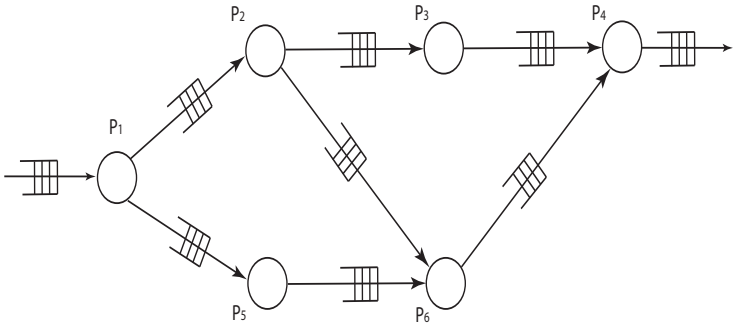


Figure 2.5: A sample KPN model

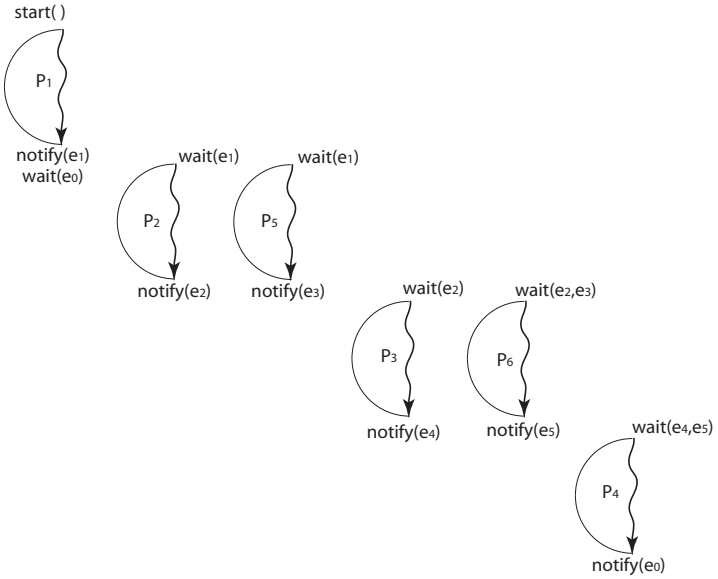


Figure 2.6: Execution of KPN model in a parallel fashion

This expression represents one of the possible implementations from the design specification as a KPN. The assumption here, is that each process needs to execute to completion to produce data for the subsequent processes. For implementation purposes, the given KPN model can be transformed into multiple threads with a deterministic order for their executions except for the processes that are completely independent (e.g., P_2 and P_5). In Figure 2.6 one multi-threaded implementation for the specification in Figure 2.5 is shown. As shown in Figure 2.6 there is a deterministic order of the thread executions except between P_2 and P_5 and P_3 and P_6. In case, sequential code is to be generated, two of the possible orders for execution are expressed as follows:

$$Order1 : [P_1 \ ; \ P_2 \ ; \ P_5 \ ; \ P_3 \ ; \ P_6 \ ; \ P_4]^*$$

$$Order2 : [P_1 \ ; \ P_5 \ ; \ P_2 \ ; \ P_6 \ ; \ P_3 \ ; \ P_4]^*$$

Correctness preserving refinements can transform a KPN model into GALS implementations [130]. A KPN model when transformed to implementation level faces problems related to buffer sizing. Several scheduling algorithms have been proposed for estimating the buffer sizes for KPN models and for validating their correct behavior [67, 115]. They are expensive and balancing rates of processes is difficult. KPN model is thus practical when coarse grain computation has to be performed with limited communication overhead.

Petri nets

Petri Nets is a specification formalism which can be used to model concurrent specifications [116]. It is a type of directed graph with nodes representing places and transitions. They are represented by circles and by vertical bars respectively. Arcs in the directed graph are from a place to a transition or from a transition to a place. A transition or an event occurs when the pre-conditions and post-conditions associated with its input and output places are satisfies. The presence of tokens in a place signifies a true output for conditions and are represented by dots within the circles [110]. Figure 2.7 (a) shows the representation of a Petri net with three places and three transitions.

With a highly concurrent Petri net model for a given computation, one could discover concurrency, synchronization points etc, much more readily when compared to standard C multi-threaded programming. However, Petri nets have their own limitations as a specification formalism. First, one could unintentionally go very close to the implementation model in the Petri net itself, by sequentializing some transitions unnecessarily, and thereby eliminating possibility of concurrency. Figure 2.7 describes one such scenario where in Net 1, there is concurrency between tasks T_1 and T_2, but in Net 2, the concurrency is eliminated by ordering them. The application being modeled in this example may have had no reason to sequentialize the two tasks T_1 and T_2, except that the modeler had made a decision to do so. Also, since Petri nets are graphical formalism, often it is hard to manage for large scale programs.

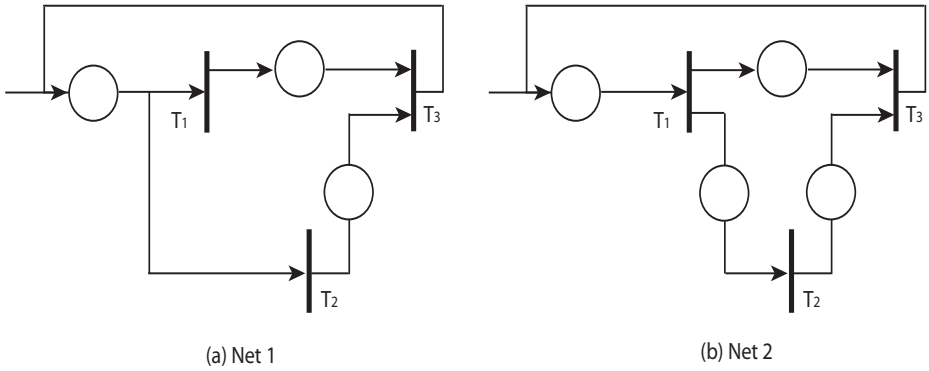


Figure 2.7: Petri net representation of sequential and parallel processes

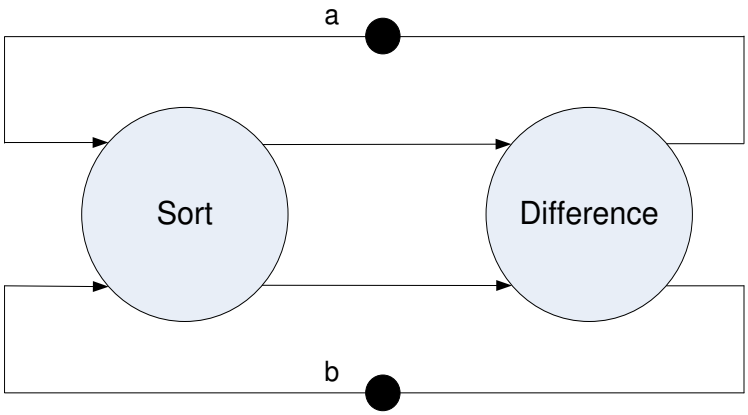


Figure 2.8: Data flow model for Euclid's greatest common divisor algorithm

Data flow networks

An alternative to Petri nets is to use the data flow model of computation. In data flow MoC (also known as synchronous data flow), the variables are considered as infinite sequences of data values, and the valuation from one step to the next is done by various operations on the data streams [104]. The MoC is expressed in terms of a network with actors representing the nodes and tokens representing the data streams. When an actor is computed (or fired), it takes in a finite number of tokens and put out another finite number of tokens. Figure 2.8 shows a simple data flow model for Euclid's greatest common divisor algorithm. The computation performed in the nodes are to sort the input data shown as tokens and to find the difference between the outputs of the sorted data.

The data flow model is very modular and is determinate. Being determinate means that the firing order of the actors can be changed without affecting the final result of the data flow graph. Once

implemented in hardware, the computation is formed using combinational logic and the communication is stabilized using buffers. In contrast with KPN, this deterministic MoC is not limited by blocking or non-blocking properties. We can calculate the buffer size with respect to the number of computations in the network at the data flow abstraction level. The number of tokens at an actor input is a condition to be satisfied for an actor to fire. There is a possibility of a deadlock condition, if the input and output rates of actors connected to each other cannot match. Also if an actor supplying tokens has a production rate higher than the consumption rate of its connected actor, buffers sizes required for implementation will become infinite. Some synchronous data flow models could have an admissible schedule where the production and consumption rates match and there will be no deadlock or infinite tokens being generated. A periodic admissible sequential schedule (PASS) is possible, where a finite sequence of firings can bring buffers back to their initial state and thus keep buffer sizes non-negative [104].

Synchronous Data Flow has been used as an MoC for sequential code generation and for hardware synthesis. The static sequential schedule obtained from PASS can be implemented as a single threaded code or as as hardware which can be improved by pipelining. For concurrent code generation, each actor can be implemented as a thread and APIs can be used for inter-thread communication. This will follow a pattern as shown in Figure 2.6. A major contribution of SDF was in promoting a new class of languages which adhered to the synchronous model of computation. The principles of synchrony, determinism, etc. present in SDF MoC was adopted to form a new class of programming languages called synchronous programming languages. Initial research focus was on sequential code generation from these languages, but concurrency was retained at the specification level which helped further work into multi-threaded code generation.

2.2 Synchronous Programming Languages

Synchronous programming languages are those languages which have synchronous execution of code as the central concept in their design. They are reactive, as each statement is executed as events arrive at inputs. At the heart of a synchronous language is the *synchrony hypothesis*. It declares that in the design of a synchronous system, an assumption about the time for computation and communication can be made. The time required for communication and computation in a synchronous system can be assumed to be instantaneous. There is an abstract notion of an *Instant* which defines the boundary for execution of statements for each reaction. This concept of Instant has no relation with the hardware clock in a circuit nor the execution clock of a processor. It is more like a marker for completing a set of actions and for deciding the next batch of statements to be executed. So by the synchrony assumption, the operations to be performed is completed within an instant and the communication is assumed to be complete before the computation begins in the following instant [101].

Concepts such as synchrony, instant, ordered events etc. are important in the understanding of this dissertation. So here we informally introduce these concepts which will be formalized in

Chapter 6. Figure 2.9 shows three signals a, b, c with events on each shown in a vertical line. For example, a has three events while b and c have two each. An event is an occurrence of a value on a signal and for one particular signal there exists a unique order of events. In other words, the events on a signal are *totally ordered*. This total order relation is shown by the continuous vertical lines on each signal. Also each of these events has a time tag which defines its instant. The dashed lines in Figure 2.9 between events a_1 and b_1, also a_3 and c_2 denote *synchronous events* or events that occurred together. So the computation on these signals were concurrent and the events which occurred on them belong to the same instant. It is interesting to note that in spite of the figure showing three events for a and two for b, we have no knowledge of whether the event b_2 occurs before, together or after a_3. The only assured fact is that b_2 happens before a_1 because a_1 belongs to the same instant as b_1 which in turn occurred before b_2. So the events on the three signals shown in Figure 2.9 is what we call as a *partial order* of events, where certain events are known to precede the other. There is no unique order for these events and hence we can have different orders of occurrence for events in this system which are called behaviors of the system. One possible behavior is $a_1, b_1, c_1; b_2; a_2; a_3, c_3$ [2]. Another behavior could be $a_1, b_1; c_1, b_2; a_2; a_3, c_2;$. So this system does not have a deterministic behavior. Once a synchronous system is specified in terms of a synchronous language, the compiler which acts on the specification may generate sequential code. So it has to form a total order on events to impose sequential order and thus a compilable synchronous program acquires deterministic output or deterministic order of execution.

The class of synchronous programming languages have four properties which are common to them, namely synchrony, reactive response, concurrency and deterministic execution. All the languages in this class are *synchronous* in their operation and executes a batch of operations within a common software clock instant. The communication between modules also follow these property by sending or reading messages instantaneously. *Reactive response* is a result of the event driven input concept of these languages. The presence of an event at an input signal triggers the evaluation of the firing condition of a synchronous statement, and may result in the execution of code. The class of synchronous programming languages have the ability to capture *concurrency* at a high level. The execution of modules or statements can be specified independent of each other. If unrelated signals are triggering mutually exclusive set of statements, the lower level code will be executed completely in parallel. This might not be true in the case of compilers which generate sequential code. Finally, *Deterministic execution* of synchronous programming languages can be guaranteed since the computation and communication are to be completed before the next instant. Given the set of input events and the synchronous program, the output of each module can be predicted for every instant.

Several synchronous programming languages have been proposed which encompasses these properties for synchrony [73]. They differ in terms of their applications, compilation schemes, specification (textual, visual) or code generation methods (sequential, parallel). Some of them have been commercialized as software tools and have found acceptance in the safety critical fields like aviation, power plants, etc. We introduce two among the most popular synchronous languages in

[2] Here events within an instant are separated by‘,’ and events between instants are separated by ‘;’.

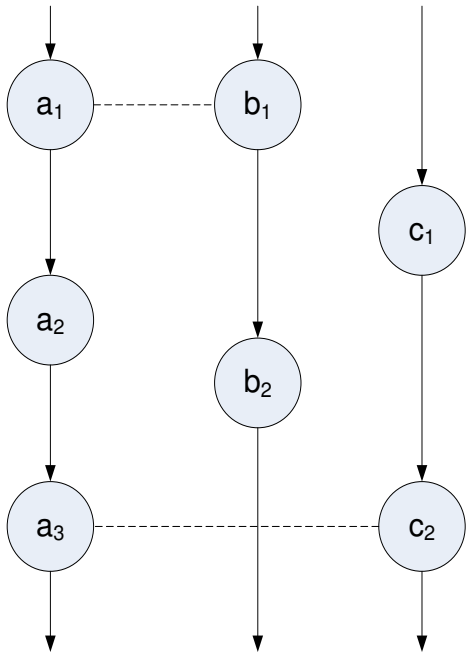


Figure 2.9: A synchronous system with three signals

this section.

2.2.1 The Imperative Synchronous Language Esterel

Esterel is an imperative synchronous programming language for the development of complex reactive systems [43, 42]. The development of the language started in the early 1980s as a project conducted at INRIA and ENSMP. Testing for presence of an event on a signal is allowed in Esterel and accordingly specific computations are selected for execution. In Esterel language, there are two types of basic objects: signals and variables. *Signals* are the means for communication and can be used as inputs or outputs for the interface or as local signals. 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. The *value* attribute of a signal is permanent and if the signal is absent, it will retain the information from previous instant. Esterel assumes instantaneous broadcasting of signals. Once a signal A is `emit` by a statement, the statements which are "listening" to this signal will be active. The scope of a signal is valid all through the module it is defined in and can be passed to another module for computation. *Variable* is local to the module it is defined in and unlike the *signal*, can be updated several times within an instant. Listing 2.2 is a sample Esterel program `ABRO` with three input signals A, B, R and one output signal O. The loop is restarted for each event on R. If there are events on

both A and B within two occurrences of R, there will be an output event on O.

Listing 2.2: A sample Esterel program [121]

```
1  main module ABRO:
2  input A, B, R;
3  output O;
4
5  loop
6     { await A || await B };
7     emit O
8  each R
9
10 end module
```

An Esterel program consists of modules, which in turn are made of declarations and statements. The declarations are used to assign data types and initial values (optional) for signals and variables. Statements consist of expressions which are built from variables, signals, constants, functions, etc. The expressions in Esterel are of three basic types, namely data, signal and delay. Data expressions are computations performed using functions, variables, constants or current value of a signal (denoted by '$?A$'). Signal expressions are boolean computations performed on the *status* of a signal. Logical primitives like `and`, `or`, `not` are used in these expressions to obtain a combinational output (eg: a `and not` b). Delay expressions are used in temporal statements along with primitives like `await`, `every`, etc. to test for presence or to assign the statements to be executed. For example, `present` A `then` $< bodyA >$ `else` $< bodynotA >$ `end` checks for the presence of A and selects between two sets of statements $bodyA$ and $bodynotA$.

Esterel expressions are converted to finite state automata with the statements as datapath and conditions as guards. The finite state machine programming model is used as the underlying formalism to convert Esterel expressions to RTL or C code during synthesis. Esterel Technologies provides a development environment called Esterel Studio [131] based on the Esterel language. Esterel Studio takes Esterel specification as input and generates C code or hardware (RTL) implementations.

The Quartz language

Quartz is an imperative synchronous language developed by the Embedded System Group at University of Kaiserslautern [124]. The syntax and semantics of Quartz are similar to Esterel. The zero time computation and communication is applicable to Quartz as well. An instant is defined by the computation time for the code within two 'pause' statements. In addition, Quartz allows synchronous concurrency where partitioning of the macro steps into micro steps having local nondeterminism is permitted. Multiple Quartz modules which synchronize at the completion of each macro step could have independent reactions inside them. These independent reactions are micro steps which are to be completed at the next synchronization point of the macro step. This feature enables multi-threaded implementations for synchronous modules while not compromising on the perfect synchrony assumption at the top module level.

Quartz programs are translated to synchronous guarded actions where a trigger condition is iden-

tified for assignments, assertions and so on [48]. When the guard condition is satisfied, the action associated with it can be executed. Delayed or immediate assignments, assertions, etc. are some of the possible actions. Quartz allows different parallel execution choices such as synchronous, asynchronous or interleaved executions and the definition of signals with infinite or finite bit width, multi-dimensional arrays.

2.2.2 The Declarative Synchronous language LUSTRE

LUSTRE is a declarative synchronous language based on data flow model [74]. In Lustre programs, every signal clock is analyzed in reference to a single simulation clock. Here, multi-rate signals have to be expressed as undersamplings of the simulation clock. The data flow approach allows the modeling to be functional and parallel, which helps in verification and safe transformation. In LUSTRE, a variable is an infinite stream of values or a *flow*. Each variable is associated with its *clock* which defines the presence or absence of the variable at an instant. The statements in LUSTRE are made of data flow equations, which result in the clock equations of the respective variables as well. There are four temporal operators in LUSTRE, namely `Pre`, `->`(followed by),`when` and `current`.
1) `Pre`(e) provides the previous value in the flow of the event e.
2) x `->` y orders sequence x followed by y.
3) $z = x$ `when` y is a sampler which passes value of x to the output z when the boolean y input is true.
4) `current`(z) is used with $z = x$ `when` y and it memorizes the last value of x for each clock instance of y.

In the LUSTRE compiler, *Clock calculus* is performed to find the clock hierarchy of the variables. Listing 2.3 gives a sample LUSTRE program for an `ALARM` which accepts an input count as $delay$ and an input trigger as set. Once an input event occurs on set, the $delay$ is loaded as $count$. A decrement operation happens until the $count$ reaches zero, which sets the output signal $level$. The decrement can be preempted as well by another event at set.

Listing 2.3: A sample LUSTRE program [76]

```
1 node ALARM(set: bool; delay: int) returns (level: bool);
2 var count: int;
3 let
4   level = (count>0);
5   count = if set then delay
6     else if false->pre(level) then pre(count)1
7     else 0;
8 tel
```

The LUSTRE language is developed by Verimag and it is the core language behind the tool SCADE from Esterel Technologies [62]. The data flow concept behind LUSTRE enables easier verification and model checking using the tool Lesar and hence is popular for modeling safety critical applications like avionics, nuclear plants etc. Esterel and Lustre are based on the synchronous data flow

model of computation with similar implementation schemes. Both follow what we call a globally synchronized model with a global software clock deciding the instants in the network. For example, the LUSTRE program ALARM has two inputs which are independent of each other. But for a LUSTRE program to compile, they have to be related to a global software clock. The inputs signals of ALARM can have events only with events of this global software clock. Esterel and LUSTRE allows the programmer to test the presence of input signals and decide on performing computation. There are several drawbacks for this approach which are illustrated using examples below.

Why use multi-rate formalism ?

A few examples which highlight the need for multi-rate formalism was presented in Chapter 1 using *Example 1.3.1* and *1.3.2*. Here we discuss some of the problems facing implementation of multi-rate signals using global software clock model.

Example 2.2.1 (Memory:)**.** *Suppose you want a system that accepts a stream of values from signal a, and outputs the same value. However, you want the system to have memory so that if its output is sampled by another system, and no event on a occurs at that sampling time, the output must provide the last output value (memorized).*

In this case, the system reads a only if either an event on a occurs or if another system is requesting this program to give an output. If a does have an event, that new value is forwarded as output. If this checking was on a request, and a does not have an event during that request, it sends the old memorized value. In single-rate driven system, at every global tick, the system must check for event on a, even when not requested, as the notion of global tick is external to the specification of the program. In a multi-rate MoC, the only time the program wakes up is when a has a new event, or when a request for an output arrives.

These examples show that having an a priori notion of global tick which is external to the specification, results in generation of inefficient code. If the inter arrival time between environment events or requests is unpredictable, a global tick driven program will be less efficient. Multi-rate formalism of SIGNAL realizes this drawback and generates code which would have lesser sampling instances as opposed to single-clock frameworks. Recently, various multi-rate extensions to Esterel have been proposed as well [44], affirming the importance of multi-rate systems in the embedded systems context. We move on to another example where the multi-rate specification based implementation is more efficient due to relative performance difference between different parts of a program.

Example 2.2.2 (Razor[35]:)**.** *Consider a system that computes a function f on two synchronized input signals a and b. In order to check that the software implementation of f is correct, one can have multiple implementations for f, say P_f and Q_f. The inputs taken from a and b are simultaneously passed to both these P_f and Q_f, and outputs are compared as shown in Figure 2.10. If the outputs are equal always, then P_f and Q_f are both implementing the same function f, otherwise not. (c.f. program checking [45]).*

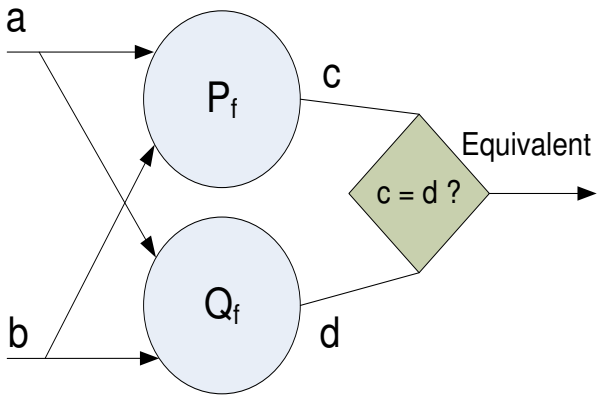


Figure 2.10: Comparison of two different implementations - Razor Example[35]

If we have a global synchronization or ticks at which inputs are fed to P_f and Q_f, by synchrony hypothesis, the outputs from both P_f and Q_f come out instantaneously and they are compared instantaneously. Now consider the case where the implementation of P_f runs much faster than that of Q_f. If the specification was in a global tick based formalism, either the inter-tick interval must be max(WCET(P_f),WCET(Q_f)), or P_f must be suspended until the tick at which Q_f completes. This requires a suspension code around P_f, watching at every tick if Q_f has completed etc. In polychronous formalism, one has to just state that P_f and Q_f must synchronize. A 'completion' signal from Q_f will be enough for this synchronization, and there is no need to check at predetermined intervals.

Using examples 1.3.1, 1.3.2, 2.2.1 and 2.2.2, we have motivated the need for a multi-rate formalism within the synchronous data flow model of computation. The synchronous language SIGNAL was conceived as a multi-rate or polychronous synchronous language to solve these outstanding issues. The programming languages Esterel and LUSTRE were later extended to implement multi-rate signals which will be discussed in Chapter 3.

2.3 The Multi-rate Synchronous language SIGNAL

SIGNAL is a declarative synchronous language that is multi-rate [71]. SIGNAL captures computation by data flow relations and by modularization of processes. The variables in this language is called as signals and they are multi-rate. This means that two signals can be of different rates and can remain unrelated throughout the program. This is a significant departure from LUSTRE data flow specifications which defines a global clock which is synchronous with every clock in the code. SIGNAL language and its Polychrony compiler has been developed by IRISA, France.

2.3.1 Basic Concepts

The SIGNAL language consists of statements written inside processes, which can be composed together. A signal x is tied to its clock $\hat{x}$ which defines the rate at which the signal gets updated. A signal can be of different data types like boolean, integer, etc. The statements inside a process can be assignment equations or clock equations. If there is no data dependency between the input signals of one statement with the output signal of another statement, they are concurrent within the process. No two signals can be repeatedly assigned to within a process. The assignment statements will consist of either function calls which are defined by other processes or any of the four primitive SIGNAL operators. They are as follows:

The *Function* operator `f` when applied on a set of signals $x_1, x_2, .., x_n$ will produce an event on the output signal y and is represented in SIGNAL as :

$$y := \texttt{f}(x_1, ..., x_n) \tag{2.1}$$

Along with the function operator the clocking requirements for the input signals are specified. To evaluate an operation on n inputs, all n inputs need to be present together and this equates the rates of y with each of the input signals. An example of an XOR operation where a Boolean signal is represented by a true (*tt*) or a false (*ff*) is shown below. $\perp$ represents an absent event and each column represents an instant.

x:	ff	$\perp$	tt	tt	$\cdots$
z:	tt	$\perp$	ff	tt	$\cdots$
x xor z:	tt	$\perp$	tt	ff	$\cdots$

The *Sampler* operator `when` is used to check the output of an input signal at the true occurrence of another input signal.

$$y := x \texttt{ when } z \tag{2.2}$$

Here z is a boolean signal whose true occurrence passes the value of x to y. The true occurrence of z is represented as $\widehat{[z]}$. The clock relation of y is defined as the intersection of the clocks of x and $[z]$. An example where v_i represents values of signals at instants $i = 1, 2, ..$ and $\perp$ represents an absent event is shown below.

x:	v_1	v_2	v_3	v_4	$\cdots$
z:	$\perp$	tt	ff	tt	$\cdots$
x when z:	$\perp$	v_2	$\perp$	v_4	$\cdots$

The *Merge* operator in SIGNAL uses `default` primitive to select between two inputs x and z to be sent as the output, with a higher priority to the first input.

$$y := x \texttt{ default } z \tag{2.3}$$

Here the input x is passed to y whenever $\hat{x}$ is true, otherwise z is passed on whenever $\hat{z}$ is true. So the clock of y is the union of the clocks of x and z. An example of a `default` operation with all four possible combinations of input events is shown below.

y:	$\perp$	v_2	v_3	v_4	$\cdots$
z:	w_1	$\perp$	w_3	$\perp$	$\cdots$
y default z:	w_1	v_2	v_3	v_4	$\cdots$

The *Delay* operator in SIGNAL sends a previous value of the input to the output with an initial value k as the first output.

$$y := x\$ \; init \; k \tag{2.4}$$

Here previous value of x, denoted by $x\$$ is sent to y with initial value of k, a constant. The clock of signals y and x are equated by this primitive. An example of `delay` operation is given below and the clock equations of the SIGNAL operators are summarized in Table 2.1.

x:	v_1	v_2	v_3
x$ init k:	k	v_1	v_2

SIGNAL operator	SIGNAL expression	Clock relation
Function	$y = f(x_1, x_2, \ldots, x_n)$	$\widehat{y} = \widehat{x_1} = \ldots = \widehat{x_n}$
Sampler	$y = x \; when \; z$	$\widehat{y} = \widehat{x} \cap \widehat{[z]}$
Merge	$y = x \; default \; z$	$\widehat{y} = \widehat{x} \cup \widehat{z}$
Delay	$y = x \; \$ \; init \; k$	$\widehat{y} = \widehat{x}$

Table 2.1: SIGNAL Operators and clock relations

The clock equations are not only inferred from the SIGNAL statements, they can be mentioned explicitly as well. For example, $\{x \; ^+ \; y, \; x \; ^* \; y, \; x \; = \; y\}$ represent the union, intersection and equate operation respectively on the clocks of x and y. Apart from these primitive operators, there are other operators like `Cell` which give a persistent output signal. The output signal will contain the values of the first input signal for all its instants and will retain the previous value of the first signal during the 'true' instances of the second Boolean input.

$$y := x \; \texttt{cell} \; zinit \; k \tag{2.5}$$

Here the events of x will be passed onto y and they will be retained when z has a 'true' output. The value k is the output if there was no initial value on x. So the clock of y is the union of the clocks of x and z. This is not a new primitive since this functionality can be implemented using the 4 primitive operators and clock constraints. An example of the cell operator is shown below.

x:	$\perp$	v_2	v_3	$\perp$	$\cdots$
z:	tt	$\perp$	ff	tt	$\cdots$
x cell z init k:	k	v_2	v_3	v_3	$\cdots$

Each SIGNAL statement in a process is a concurrent unless there is a data dependency with another statement within the process. There can be variables which are internal to the process ($P \ / \ x$). The processes can be composed similar to the manner in which each SIGNAL statements are composed ($P \mid Q$). Now let us consider a sample SIGNAL program `Count` for a counting process with reset. It accepts an input event reset and delivers the integer output val. A local counter, initialized to 0, stores the previous value of val (equation counter := val$1 init 0). When the event reset occurs, val is reset to 0 (i.e. `(0 when reset)`. Otherwise, counter is incremented (i.e. `(counter + 1)`). The activity of Count is governed by the clock of its output val, which differs from that of its input reset: Count is multi-clocked.

```
process Count= (? event reset ! integer val)
   (|  counter := val$1 init 0
    |  val     := (0 when reset)
                  default (counter + 1)
    |) where integer counter;
end;
```

The process is named as `Count` and the input-output ports are declared using the symbols '?' and '!' respectively. Each input or output is associated with its type event, integer, Boolean etc. Each SIGNAL process is a synchronous composition of concurrent statements separated by the '|' symbol. Similarly, processes can be composed if they are concurrent in nature. Each SIGNAL statement consists of the 4 operators used with input or output signals or internal variables like $counter$ in this example. With the familiarity gained from the definition of operators and examples, we move onto explaining charateristics of SIGNAL programs such as endochrony and weak endochrony.

2.3.2 Endochrony and weak endochrony

The properties endochrony and weak endochrony were not introduced along with SIGNAL language, rather were identified as a conditional property for compiling programs written in SIGNAL. The formal semantics and notations needed to precisely define these two concepts is quite elaborate [72, 119, 39]. We provide examples to clarify the concepts, hoping the reader will glean the conceptual idea behind these definitions.

Definition 2.3.1. *A SIGNAL process P with input signals $i_1, i_2, ..i_n$ and output signals $o_1, o_2, ..., o_m$ is endochronous if and only if the scheduling of the computation based on the events arriving at the input signals can be correctly inferred without any additional information from the environment during the run-time of the process. That means, a correct scheduling of the internal computations to produce the correct sequence of output events can be statically done at the compile time.*

Let us consider two simple examples to illustrate the issue.

```
process Ex1 = (? integer x, y ! integer val)
   (|  val := x + z$1 init 0
    |  z     := (y + 1)
    |) where integer z;
end;

process Ex2 = (? integer x, y ! integer val)
   (|  val := x + z$1 init 0
    |  z     := (y + 1) default 0
    |) where integer z;
end;
```

In the process Ex1, the two inputs x and y are synchronous, because from the body of the computation it is clear that x and z are present at the same instants always, otherwise, the computation of val would not be possible. The second equation tells that z and y must occur at the same instants, and hence by transitivity, events on x, y are always expected from the environment at the same instants. Therefore, the scheduling of the computation can be done statically. First read events on x and y together, then produce the output event on val immediately, and then go back to reading the values on x, y again. So Ex1 is an example of an endochronous process. Generating sequential C code for such a process is straight forward.

In the process Ex2, however, the second equation does not imply that events on z occur always at the same instants as y events, because if y events are absent, z can still have events. Hence, even though from the first equation events x and z are synchronously occurring, it cannot be inferred that x, y are synchronous. This means, reading input event on x does not mean reading input event on y, and therefore, the value of val would be dependent on whether y occurred just before the new event on x is read in. So this computation cannot be statically scheduled, without getting information from the environment on when when x, y occur simultaneously and when not. So Ex2 is an example of a non-endochronous process. Generating sequential C code for this case is not possible, without compromising the intended semantics. Now we move onto another class of SIGNAL programs which are not implementable in current SIGNAL compilers due to non-deterministic behavior.

Definition 2.3.2. *A SIGNAL process P is weakly endochronous, if and only if it has the 'diamond property' [114, 119]. In other words, for such a process, even when a complete static schedule of the computations is not possible, order of the computations may dynamically depend on the occurrences of events on the input signals, the computation is confluent. The confluence here means that irrespective of the dynamic order of the computations, the final state of the process is the same as the case where all input events occurred synchronously.*

The following example will make it easier to understand the concept of weak endochrony and its diamond property.

```
process Ex3 = (? event a, b;
! integer val1, val2;)
   (|  val1 := (x + 1 when a)
   (|  val2 := (z + 1 when b)
    |  x    := val1$1  init  0
    |  z    := val2$1  init  0
    |) where integer x,z;
end;
```

In process Ex3, one cannot determine the relative rates of a, b and hence a static schedule of when to do computation is not possible. However, if a occurs before b, or vice versa, or even if they occur together, the value produced after every pair of a and b will be the same. In fact, every so often, the events on $val1, val2$ will be the same for a particular occurrence rate of a, b, and a fully synchronous occurrence of a, b events. Note that one cannot generate sequential C code for this process, because if the code waits for events on a, and if an event on b occurs first, the event on b will be lost without being read. The same is true if the sequential code tries to read events on b first.

A weak endochronous program will have the *diamond property* which shows the multiple behaviors possible according to the relative modification of values of each output signal. Figure 2.11 shows the diamond property for the SIGNAL program `Ex3`. Here we have three cases, where a and b events happen together or one of them happens before the other. The initial values of output signals $val1,\ val2$ is $x,\ z$. Once an event occurs on a, $val1$ is updated to $x+1$. Similarly $val2$ will change to $z+1$ if an event occurs on z. Another possible behavior is $val1$ being updated after $val2$ or both being updated together repsented by different branches of Figure 2.11. These different behaviors are against the determinism required by SIGNAL compilers for sequential code generation. We investigate the multi-threaded code generation opportunities on weak endochronous SIGNAL specifications to broaden the class of implementable SIGNAL designs in Chapter 5.

Endochrony and Weak endochrony are properties of synchronous system pertaining to the schedulability of their computation. They help decide whether a specification has deterministic order of execution which is required by compilers for sequential code synthesis. Another property equally important while composing IPs is *isochrony*. Isochrony is condition which applies for variables shared between IPs where they have to agree on values assigned to them at each instant [129]. There should not be reactions having contradictory values for shared signals. Isochrony has to be viewed in relation with properties like endochrony and weak-endochrony, since the compositionality of two endochronous processes or two weakly endochronous processes require different assurances. In Chapter 4, we study the compositionality of synchronous systems for each of these cases.

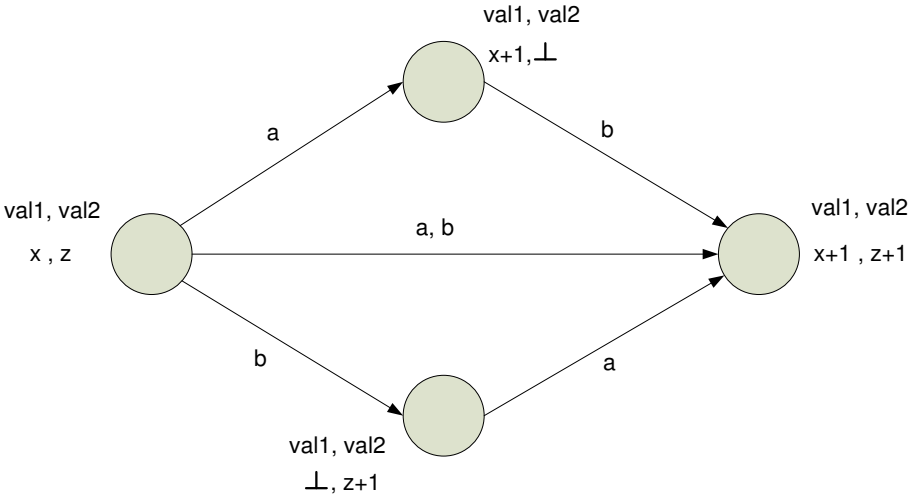


Figure 2.11: Weak Endochrony or Diamond Property for the SIGNAL program Ex3

Chapter 3

Related Work

In this chapter, we discuss the important contributions in the field of code generation and parallel programming. The background information about the parallel programming basics and the different models of computation were given in Chapter 2. Here we concentrate on the software synthesis tools, their methodologies and capabilities to implement designs in the form of C or RTL. In particular, we discuss how sequential and multi-threaded software synthesis is made possible.

3.1 Parallel programming research

The emergence of multi-core systems have necessitated the conversion of existing sequential programs into a parallel form for faster execution. The distributed memory models have taken a back seat in handling this task, while synchronization objects defined in POSIX/Windows threads are too low level to tackle this problem. Higher level APIs like OpenMP and Thread Building Blocks (TBB) have been proposed for this purpose using shared memory based models. Specialized pragmas defined in OpenMP and TBB are associated with loops or any other places which need parallelization. These models of parallel programming are non-invasive, since they can be ignored in an environment that does not support these pragmas.

3.1.1 OpenMP

OpenMP [19] is a set of compiler directives, runtime routines and environment variables used to express parallelism in code. They can be Fortran directives or C/C++ pragmas ($pragma\ omp$) , which alter the control flow into a fork-join pattern. When encountered with a *parallel* construct followed by an iterative loop, the compiler will create a set of slave threads which divide the iterations among themselves and execute them concurrently. The number of threads can be set by the user by means of an OpenMP directive $omp\_set\_num\_threads()$ or by setting a related environ-

ment variable *OMP_NUM_THREAD*. The type and scope of the data structures of each thread is important when used in a multi-threaded context. A variable can be explicitly specified as a *shared* or *private* variable under threaded conditions using OpenMP directives. An OpenMP pragma in C/C++ can specify a set of variables to be shared within a class or structure (default case), which would ensure a single copy is maintained for those variables. For an iterative loop, the iterative index is considered as private for threading purposes. Apart from the index, any variable which will undergo update operation within a loop must also be specified as private to have separate copies for each thread. There is also a reduction operation which functions by making a combination of shared and private variables. Reduction operation makes use of the commutative-associative properties to form intermediate results and thus parallelize the operation. This is different from parallelizing the iterations in a loop as the intermediate results are accumulated to get the final result.

Parallel programming constructs are utilized to increase the performance and to ensure correctness of code. There might be statements which should not be executed in parallel or variables whose additional copies should not be made. This can be guaranteed by using *atomic* directive which halts all parallel operation for the concerned statement. *Critical - end critical* directive serves the same purpose for a section of code. There exists *set lock - unset lock* directives similar to Pthread mutex variables for providing exclusive access to variables. In comparison with Pthread constructs, a disadvantage is that the critical section in OpenMP stalls any other critical operation. Even two independent critical sections without common shared variables run in separate threads cannot be executed in parallel. Other mutual exclusion primitives include event synchronization and memory access ordering pragmas. A *barrier* directive can be used to synchronize all threads at a point which acts as a location to halt, join and proceed. The threads which finish execution will wait for others to reach the synchronization point. The *ordered* pragma provides exclusive access to memory by sequentializing a portion of code. This enables the code to perform parallel computation and sequential storing operation.

3.1.2 Thread Building Blocks

Following the lead of OpenMP, new libraries have been proposed for extending parallelization constructs to C++. Thread Building Blocks (TBB) [6] is an effort from Intel Corporation to provide shared memory parallelism in C++ with automatic scheduling of work. It aims to provide better load balancing by using task based programming instead of lower level threads. The TBB libraries can be used to perform loop parallelization, sorting, scanning etc. which we discuss in this section.

There are two major loop parallelization templates from TBB, namely $parallel\_for$ and $parallel\_reduce$. An iterative loop which can be safely parallelized can be done by using $parallel\_for$ function. The parameters for this function are the datatype, grainsize, number of iterations and the operator function to be parallelized. Grainsize describes the size of operations in each parallel processing thread and can be optimized experimentally. The $parallel\_reduce$ function performs computation in a split-join fashion. A reduction operation is performed by partitioning a long serial operation into

smaller independent parts which are merged after the computation. The distribution of computation is shown in Figure 3.1 for both $parallel\_for$ and $parallel\_reduce$ functions. The iterations are parallelized in the first case while the sub-blocks are parallelized in the second. Specialized functions like $parallel\_scan$, $parallel\_sort$ etc. are available to exploit concurrency in parallel algorithms. For memory accesses, *containers* or FIFO-like arrangement for multiple threads and standard templates for mutual exclusion are provided.

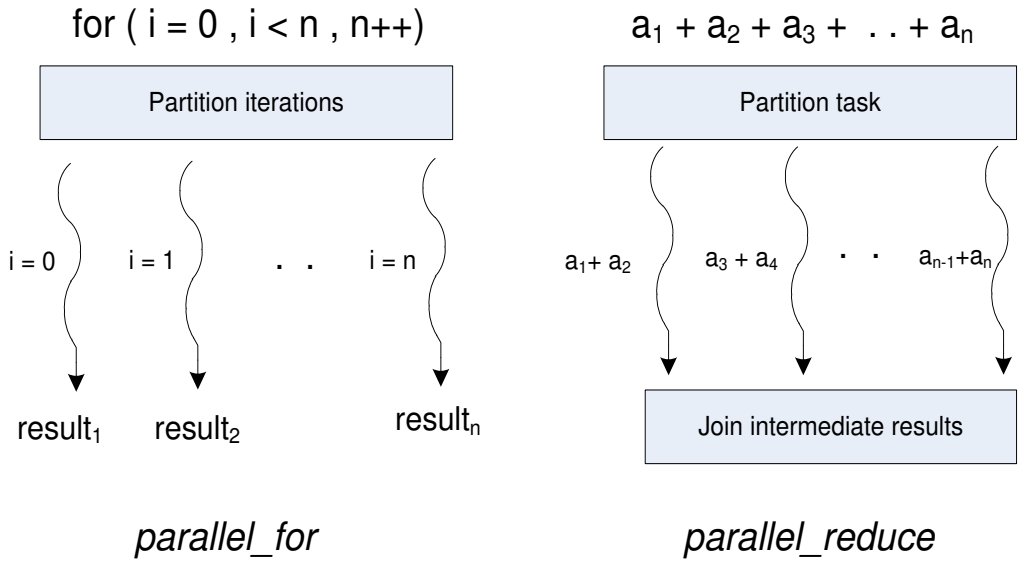


Figure 3.1: Parallel functions in Thread Building Blocks

TBB was designed to remain strictly as a C++ library to support parallelism. Compiler support was required for OpenMP, which is avoided in the case of TBB. TBB also provides nested parallelism support for more parallel algorithms. When compared to native threading, TBB influences the scheduling by providing an unfair distribution of processor execution time for each thread. Execution time is allotted based on the load on each thread and thus TBB provides better performance than other shared memory parallelism techniques.

3.1.3 Message Passing Interface

Message Passing Interface (MPI) is a programming model targeting distributed execution in multiprocessors [70]. MPI programming model consists of parallel processes communicating with each other in point-to-point fashion. In contrast to the forking of threads in OpenMP, MPI is concurrent from the very beginning. Parallel processes execute in a MIMD-like model and operate on memory with exclusive access. The focus of MPI programming model is on task division, thus reducing the communication between processes.

MPI provides several specialized functions to communicate between processes. The message can

be sent or received in a blocking or non-blocking fashion. The message passing functions like MPI_Send, MPI_Recv will contain parameters which give the starting address, size and data type of the data being sent/received along with message identifiers and communication handles. The communication handle describes the processes in a group which are allowed to receive the message. Different MPI functions provide facilities to broadcast, distribute or accumulate data within groups. Groups contain an ordered set of processes uniquely defined by their rank. The communication between these processes is termed as *intra-group communication*. It is possible to have message passing between processes that are part of separate groups. In such an *inter-group communication* environment, the identifiers for a process is the communicator (group identifier) and the rank of a process. Apart from the blocking message passing functions, specific synchronization functions ($MPI\_Barrier$) are also provided to co-ordinate the communicating processes.

3.1.4 Parallel Programming on Multi-Processors

As we have discussed before, parallel programming research started off with multi-processor systems. Programming methodologies applied in that era are the inspiration behind many of new software models applied for multi-core chips. Some of the significant multi-core processors are CellBE [2] and Sun Niagara [17]. CellBE is a heterogeneous processor with IBM PowerPC and several vector processing elements where a master-slave programming approach is used. Niagara is a homogeneous processor with eight SPARC cores, with uniform distribution of work. CellBE has found commercial success in gaming consoles and we will discuss the architecture with an associated programming model of this design in brief.

The Cell Broadband Engine Architecture [2] consists of an IBM PowerPC processor as the Power Processing Element (PPE) with eight vector processors as the Synergistic Processing Elements (SPE). They are interconnected by an Element Interconnect Bus. The PPE acts as a controller for the SPEs by performing scheduling operations, resource management and other OS services. The Cell processor can support two hardware threads in its PPE and eight hardware threads in its SPEs. But the programming model of the Cell processor is not restricted to a singular methodology. Users are free to create software threads and manage the communication between them in shared memory or message passing model. The Cell Processor supports OpenMP libraries and is flexible enough to perform multi-threading operations in pipeline, job queue or streaming format. Cell Superscalar [37] is one of the applicable programing models for the Cell processor which uses annotations to delegate tasks from the PPE to the SPEs. The PPE contains a master thread which maintains a data dependency graph for the tasks to be performed and a helper thread which schedules tasks for each SPE thread and updates the task dependency graph. The creative freedom present in the applicable programming models for the Cell processor has made it a versatile platform in the multi-core embedded system domain.

3.1.5 Parallel programming using Graphics Processors

Graphics processors have been used to offload vector processing from CPUs for a long time now. Recent advances in gaming technology has motivated researchers to look at using graphics processors for general purpose computation. The idea is to make use of the large amount of multi-processors in graphics cards to create a massively parallel system for computation. Using general purpose graphics processing units (GPGPU) for parallel processing delivers a favorable performance-cost metric when compared to the available supercomputing options. The programming of such graphics processors is very different from other embedded systems as they follow a Single Instruction Multiple Data pattern. We discuss the programming philosophy of CUDA, a leading software architecture from NVIDIA Corporation in this section.

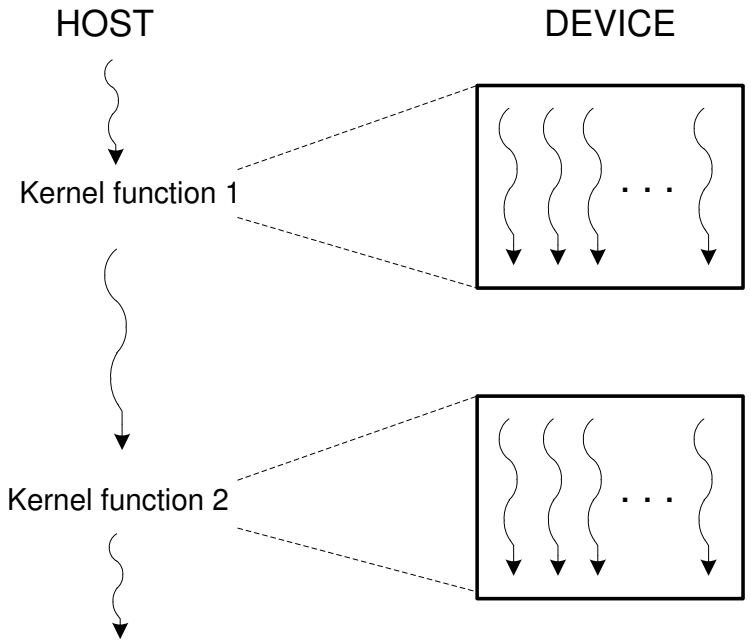


Figure 3.2: Program flow in Host and Device for NVIDIA CUDA

Compute Unified Device Architecture (CUDA) is a new programming model defined for NVIDIA GPGPUs [12]. In this programming model, the CPU (Host) code sets the amount of threads that is to be created in the GPU (Device) using a Kernel function. Each parallel operation is a Kernel function call which halts the Host code execution in the CPU and starts a massively parallel operation in the GPU. In the GPU, a set of threads are tied together to form a *warp* and is assigned to a set of processors. The warps assigned to a multiprocessor take turns in execution, memory fetch etc. Figure 3.2 shows the execution model of NVIDIA Tesla GPU. The Host code is executed in sequential fashion with pauses during the parallel Device operation. This shared programming model is scalable for larger applications. There exists specialized registers and shared memory for groups of processing units along with a global memory. Several atomic functions like

$atomicAdd(),\ atomicInc,\ atomicAnd$ are provided for safe threading operations.

The programming model used in GPUs is of Single Program Multiple Data (SPMD) pattern. Streaming data for video rendering was ideal for this model since the same computations were done for multiple pixels. Other applications of this programming model range from speeding up time consuming simulations for hardware design to movie animation [112]. Brook for GPU, a streaming model for GPU general purpose computation similar to CUDA was proposed from Stanford University [16]. The programming language is ANSI C with extensions to declare streams and kernel functions to operate on them. Here, the extended C code is transformed to an executable form for graphics processors and no new programming language is required. At a higher level of abstraction is model driven code generation which has sound formal basis in its specification format. Streaming model, data flow model, etc. have been used as a reference to design high level languages which are transformed into C or RTL using different code generation tools. A recent work has

3.2 Software synthesis from synchronous languages

Synchronous programming languages such as Esterel[43], SIGNAL[71], LUSTRE[74] and different variants of them have proved to be successful in embedded reactive software specification and generation of sequential software code from them [40]. Even though broadly classified as synchronous languages due to the underlying synchrony assumption, these languages often have very different characteristics and styles. In this section, we discuss code synthesis from two globally clocked synchronous languages, Esterl and LUSTRE.

3.2.1 Esterel software synthesis

Esterel is an imperative language which reacts to events and inputs in tight loops whose execution is often governed by traps, aborts and other kinds of exception watching. Usually Esterel programs also have the notion of a timer tick, which paces the reading of inputs and the corresponding reaction [60]. So a reaction in Esterel happens between two consecutive ticks, based on the input events and values, and often a reaction may be aborted or preempted by exception conditions.

Esterel expressions are converted into a finite automaton and synthesis is performed to generate sequential code [43]. An input automaton at state P when in the presence of an input event i, generates an output event o and moves into a derivative state P'. In this manner, a Finite State Machine (FSM) can be formed which produces a deterministic sequential output from a concurrent specification. The datapath of the FSM at each state will include the code that has to be executed at each instant. Esterel compiler can generate C code or RTL from this finite state automata. Esterel Studio is a commercial complier for Esterel langauge developed initially by Esterel-Technologies and now owned by Synfora Inc [131].

An academic Esterel Compiler is also available from Columbia University [3]. The Columbia Esterel Compiler [60] has implemented a few code generation techniques to form C code from Esterel. One method divides the code into atomic tasks and performs aggressive scheduling operations. Another method is to form a linked list of the tasks by finding their dependencies. Here the focus is on fine grained parallelism as in Object Code method [52] which is explained later in Section 3.2.3. A distributed implementation on multiprocessors [139] uses a Graph Code format proposed in [117] to represent parallelism in Esterel. Here each thread is a distinct automaton (or a reactive sub-machine). Instead of scheduling tasks during runtime as in other techniques, each sub-machine is assigned to a processor core and they are composed together to form the main-machine which represents the whole Esterel code.

Quartz software synthesis

The imperative synchronous language Quartz has its associated framework Averest for capturing specifications and for synthesizing C or RTL code from them [125]. Quartz statements are converted to synchronous guarded actions, where a guard acts as a firing condition to perform an action. Averest Intermediate Format (AIF) is an XML format used to represent intricate details of a user given program for translation into low level implementation code. From AIF format, Averest can generate code, provide verification collaterals such as smv input files or deterministic automata.

3.2.2 LUSTRE software synthesis

LUSTRE is a data flow language which describes a system as data flow equations with an assumption that there is a global notion of a time trigger that allows these data flow equations to be evaluated repeatedly, and thereby result in intermediate or output events and values. Each of these repetitions of evaluations is a reaction in LUSTRE. LUSTRE has been adopted as a specification language for code synthesis targeting many safety-critical applications. A commercial LUSTRE compiler is available from Esterel-Technologies Inc. as a part of their SCADE suite [62]. Also available is a higher order functional variant of Lustre known as Lucid Synchrone [53].

The LUSTRE compiler can also generate output in Object Code form [52] which can be used for distributed implementation as described in the Section 3.2.3. Another work on multi-processor implementation is based on Time Triggered Architectures (TTA). SCADE can be used to map LUSTRE specifications on the synchronous bus by having some extensions on the LUSTRE code in the form of annotations [51]. Code distribution annotations are used to assign parts of LUSTRE program to a unique locations in the distributed platform. Execution time, period and deadlines can also be specified along with the code. The methodology for implementation of LUSTRE program in a TTA is shown in Figure 3.3. The LUSTRE specification is given to the Analyzer which builds a partial order of tasks with the help of the deadline and execution time annotations. The timing details are used by the scheduler to solve a multi-period, multi-processor scheduling

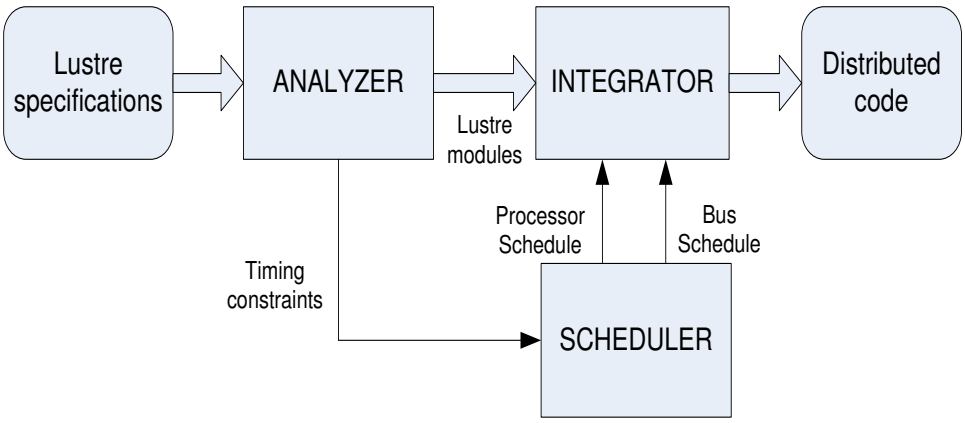


Figure 3.3: LUSTRE to TTA implementation flow

problem. The bus and processor schedules for a solution to this problem are given to the Integrator block. Integrator obtains the different LUSTRE modules from Analyzer and generates a glue code to interface these modules.

3.2.3 Distributed code generation from Synchronous languages

The underlying concurrency in the specification makes synchronous languages a good candidate for distributed implementation. A work on automatic distribution of synchronous programs proposed a common algorithm for conversion of an **Object Code** (OC) into a distributed network of processors [52]. Esterel, LUSTRE compilers can output code in this common format. The distribution method from the OC form is as follows:
1. The centralized Object Code is duplicated for each location
2. Decision is made on mapping each instruction to a unique location and copies are removed from the rest of the locations
3. Analysis is performed to find the communication required between locations to maintain the data dependency between instructions
4. New instructions are inserted (put, get) to pass the variables that were computed in a different location

Optimization can be performed to reduce the redundant code in the network. A sample Object Code is shown in Listing 3.1 and its distributed implementation for two locations is shown in Figure 3.4. The code is first duplicated on both locations and then the body of the code is removed from one of them. Later the communication instructions (put(a,0), get(1)) are placed in the locations as required. In Figure 3.4, on a *true* result on the '`If` a' condition, $body1$ is executed on `Loc0` and `Loc1` remains idle. On a *false* result, `Loc1` computes $body2$ and sends the value of a to `Loc0`. In `Loc0`, a `get` operation is performed to update the latest value of a and then $body3$ is executed.

Listing 3.1: Object Code for an if-else condition

```
1  Location     State 0
2  0            put_void(1);
3  1            put(0,a);
4  0,1          if (a) then
5  1                 put(0,a);
6  0                 body1
7  0                 output(b);
8  0,1          else
9  1                 body2
10 1                 put(0,a);
11 0                 body3
12 0                 output(b);
13 0,1          end if
14 0,1          go to State1
```

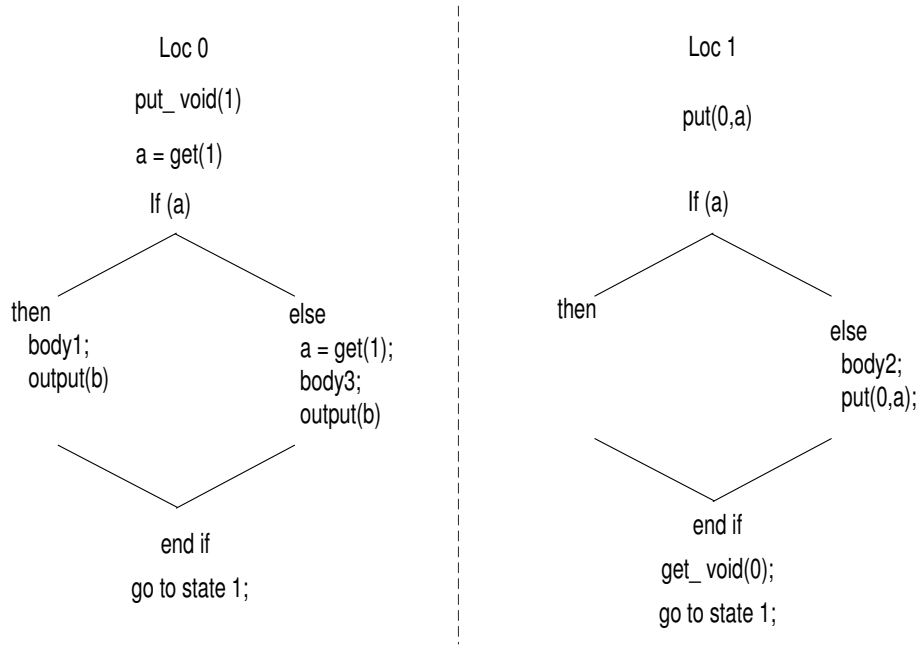


Figure 3.4: OC program in Listing 3.1 distributed into two locations

3.3 Software synthesis from Polychronous specifications

For a given polychronous (or multi-clock) specification there may not exist a unique behavior or a fixed order of computation. This is obtained by establishing a total order on all the instants of signals in the polychronous specification. With additional synchronization, it maybe possible to force a unique behavior and thus be made deterministic. We now explain the issues in sequential synthesis using a sample specification. Later we will review the available synthesis tools and their causality analysis techniques. Finally we review some of the multi-threading work performed in this area.

3.3.1 Issues in sequential software synthesis from polychronous specification

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 [71]. 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. 3.5 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. 3.5, 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.

Now, consider the polychronous specification $Spec2$ in Fig. 3.5, 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 clock of r contains all instants of a and all instants of b. A possible sequential implementation is shown in Fig. 3.5 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.

3.3.2 Synthesis tools for polychronous specification

The major synchronous programming languages like Esterel[43], LUSTRE[74] and SIGNAL[71] and their distinct ways of reacting to events have been discussed in Chapter 2. Among synchronous languages, SIGNAL [66] is polychronous in nature. A commercial version of SIGNAL based code generation tool is available as RT-Builder (Sildex) from Geensys [15]. **Polychrony** compiler [61] built at IRISA/INRIA France is the point of origin for this commercial product. It provides a frame-

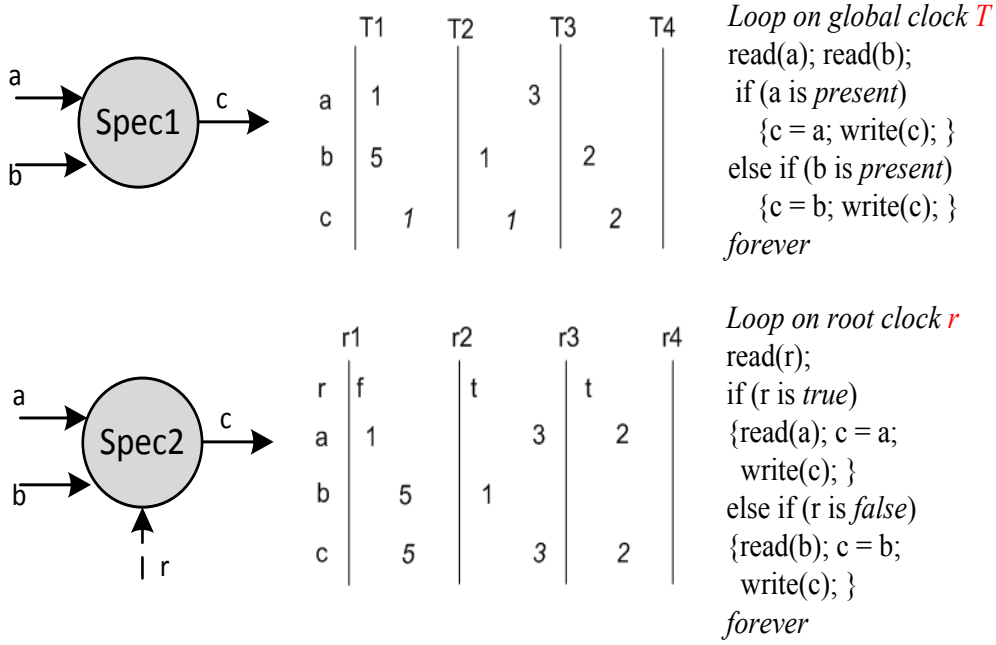


Figure 3.5: Non-endochronous and endochronous polychronous specification

work for multi-rate data flow specifications and is the inspiration for our MRICDF framework. Polychrony uses ROBDDs for creating canonical form for all clock variables, and then constructs a clock hierarchy in the form of tree-like data structures. Apart from the absence of causal cycles, endochrony property is necessary for compilation. This property at clock tree abstraction level is interpreted as the 'existence of a root clock' in the clock tree. This would mean that there has to be a signal or a combination of several signals defined within the specification or a combination of signals which can act as the fastest signal. Non endochronous and weak endochronous programs are rejected.

Polychrony environment takes textual commands through its interface along with SIGNAL code in a '.SIG' file and generates C code which can be simulated with inputs given in '.txt' files. Lack of a visual means of specification and absence of debugging facilities are among the problems with Polychrony toolset. SIGNAL-META [50] is an effort to build a front end for polychrony using the Generic Modeling Environment [108]. A meta-model close to SIGNAL was constructed with the constructs divided into three parts: the interface, the data flow and the clock relations. This tool converts the visual specification into UML-like intermediate format and finally into '.SIG' and '.txt' format. From this point the Polychrony tool will be used to generate C code. So there was no modification in terms of the compilation of SIGNAL nor in terms of testing for endochrony/weak endochrony properties.

Separate or modular compilation of SIGNAL specification into separate C programs which would

run on distributed architectures have been investigated [38, 114]. This body of work has mainly focused on characterization of SIGNAL specifications which can be separately compiled and/or can be run on a distributed architecture. Often this includes synthesis of special protocols between the different components so that the distributed system produces the same behavior as the original synchronous specification. One such characterization is *endo-isochrony* [38, 39]. An endo-isochronous specification is one whose computation is endochronous and communication is isochronous. Benviniste et. al [38] proved that a synchronous system with the property of endo-isochrony can be implemented as a semantically equivalent GALS architecture without the need for any protocol.

More recently, a more generic characterizations have been proposed such as weak endochrony [119], and NRSA (No Reaction to Signal Absence)[120]. While endo-isochrony provides a sufficient condition for the separate compilation to work, in [114] it has been shown how to extend the scope of this to weakly endochronous components. Since weak endochrony is compositional, it provides a way of incrementally compiling and synthesizing controller for resulting C programs to work correctly. In an alternate paper [33], synchronous distribution of SIGNAL programs was investigated and the SIGNAL dependencies and scheduling relations were retained through refinements until the final stage of distributed code generation.

3.3.3 Causality analysis techniques from polychronous specifications

A recent work [132] has summarized sequential synthesis condition of SIGNAL programs as follows: a compilable and hierarchic SIGNAL program is *endochronous*. In other words, a sequentially implementable SIGNAL specification would not contain causal loops and its clock tree would be hierarchic with a root clock. Over the years, there has been a significant amount of work in detecting causal loops for hardware circuits and synchronous systems. An early work on the analysis of cycles in combinatorial circuits uses ternary symbolic simulation method [107]. This work was extended later to sequential circuits and to Esterel language. 'Constructivity' [41] was proposed, which considers a circuit to be acyclic if and only if for every external input a unique value can be determined for each internal and output signal. A sequential circuit is constructive if the combinational part is constructive for given input values and the latch outputs are restricted to reachable states [127]. A heuristic was proposed where ternary symbolic simulation is performed to see if the set of unstable states is empty. If so, the circuit is declared constructive. Otherwise, a reachability analysis is done to see if any of the unstable states is reachable from a sequence of given inputs.

Checking constructivity as a Satisfiability problem was proposed in [111]. In their formal verification system, a set of internal signals are identified, which would eliminate all syntactic cycles. A constructivity-SAT formula for a cycle is defined to perform reachability analysis. In addition, an error path from initial state to a state where the cycle is true is determined. Thus the cycle analysis has been extended to finite non-Boolean types. In Esterel, instantaneous termination due to incorrect programs or schizophrenic programs was analyzed in [134]. A program is declared *instantaneous* iff the execution completes in a single reaction. The analysis of causal loops using

using SAT techniques was suggested, but was not experimented with. Some provable correct Esterel programs were noted to be rejected due to incomplete causality analysis. Other prominent works on causality analysis provide optimizations for causality elimination [123] and applying standard logic synthesis techniques [126]. In SIGNAL, a recent work [65] has experimented dead code detection based on computation of intervals for values of signals. Parts of a program can be rejected if the expected values did not fall in the computed interval. A common thread in these works is isolation of specific properties useful for causality analysis in the respective synchronous languages or the use of verification techniques such as model checking [55, 122].

3.3.4 Multi-processor and multi-threaded programming models for SIGNAL

An early work on distributed implementation of SIGNAL programs [105] discusses combining SIGNAL with SYNDEX (SYNchronous Distributive EXecutive) CAD tool [69]. SYNDEX tool can provide rapid prototyping and optimization of real time embedded applications on multiprocessors and it is based on a **potential parallelism** theory. In this method, parallelism will be exploited only if the hardware resources for parallel execution is available. A SIGNAL-SYNDEX translation strategy is defined by using an intermediate representation compatible with both languages. A Synchronous Flow Graph is later formed which is a 5-tuple with nodes, clocking constraints, precedence constraints etc. as its elements. Once the equivalent graph for SIGNAL program is constructed in SYNDEX, a clustering and scheduling strategy is applied to obtain the optimized real-time mapping onto a distributed system. A clustering phase is used to increase granularity, thus reducing the complexity of the scheduling problem into multiple processors. Once the clustering phase is done, mapping of u clusters onto p processors ($u \geq p$) is done using the SYNDEX tool. Once the virtual processors are mapped to physical processors, clusters are formed within each processing element and an efficient static schedule is found for each cluster. Meanwhile the resultant sequence in each cluster is dynamically scheduled according to the arrival of input events.

The current Polychrony compiler for SIGNAL from IRISA has implemented a multi-threaded code generation scheme. The strategy here is to use semaphores and event-notify scheme to synchronize the communication between threads. Each concurrent statement in SIGNAL is translated into a thread with a *wait* for every input at the beginning and a *notify* for every output in the end. This **micro-threading model** for a SIGNAL code is shown in Figure 3.6. Here there is a $controller$ which ticks according to the endochronous SIGNAL global clock which is a superset of all the input events. The $controller$ thread notifies the read operation for a particular input and the respective threads associated with the inputs are triggered. For example, p is computed using inputs x, z and the computation will be triggered by events $e4$ and $e6$. The semaphore wait and notify statements provide the synchronization between the threads. The multi-threading model of Polychrony is modeled to be reactive to the input and will aggressively schedule computations whenever they are available. But at the same time, the fine grained nature of the tool results in

more communication and less computation for a small task. When applied to larger SIGNAL programs, the number of threads increase exponentially since each concurrent statement in the code is forked out as a thread.

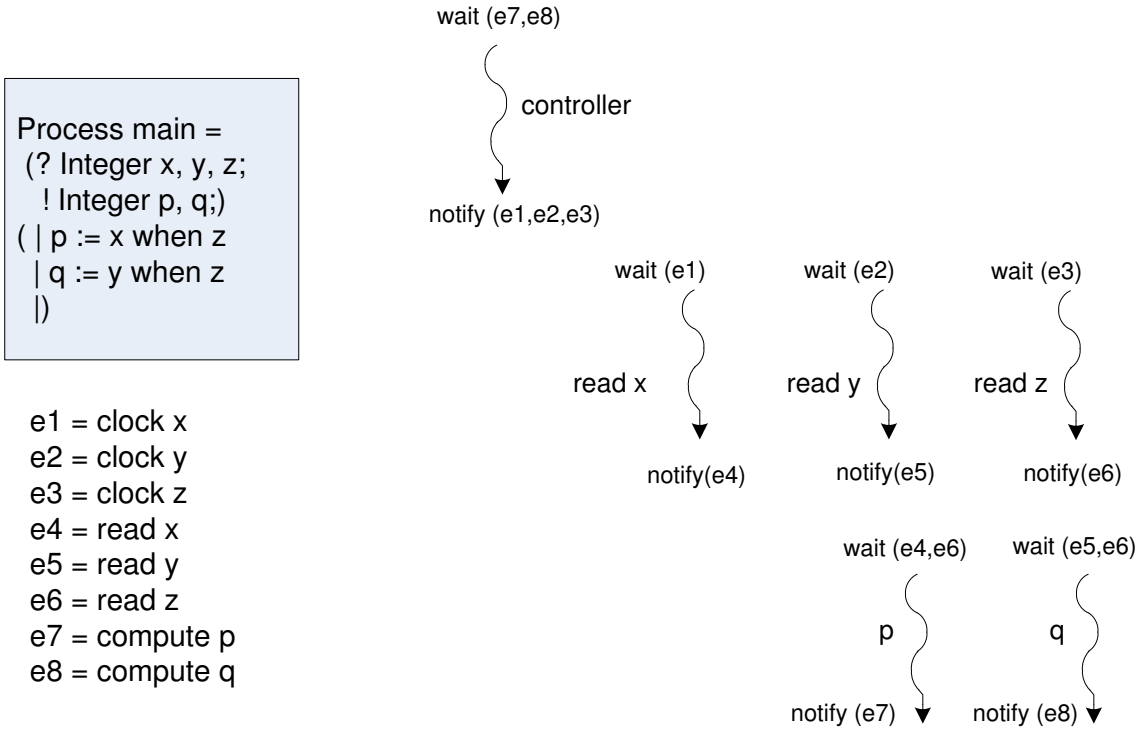


Figure 3.6: Fine grained thread structure of Polychrony

3.4 Code generation tools in industry and academia

Real-Time Workshop: Real-Time Workshop from Mathworks is a tool for development, testing and generation of embedded C programs or HDL [7]. The models can be Simulink designs or Embedded MATLAB code. They also provide simulation environment and analysis capabilities for testing the generated code. Simulink Stateflow can also be interfaced with the Real-Time Workshop for modeling finite state machine based designs. Stateflow formalism is similar to Statecharts [77], where parallel states can be modeled, thereby protecting concurrency at the higher abstraction level. At the implementation level, a sequential order is chosen for generating sequential C code.

MATRIXx: MATRIXx is a set of design and development tools from National Instruments

(NI) for graphical system modeling, simulation and code generation for embedded systems [11]. AutoCode is their real-time code generation tool for single-rate and multi-rate systems. But, here multi-rate signals are globally synchronized and the rates of individual signals are related to the global clock signal. A high-level model designed and simulated using Xmath and SystemBuild tools in the MATRIXx environment is later given as the input to the AutoCode tool.The target languages of AutoCode is C or ADA.

StreamIt: StreamIt is a programming language which is used to model designs which handle streaming flow of data [136]. Each stream of data is acted on by blocks called 'filters'. Filters can be pipelines, feedback-loops or split-joins which can be used to fork out threads of execution. The data flow structure makes StreamIt more suitable for multi-core execution. Parallel threads which access a shared memory and communicate using sockets can be generated using StreamIt compiler. Code generation into C language format from StreamIt models targets multi-media and signal processing applications.

Ptolemy: In academia, Ptolemy is a significant contribution for actor based networks for code generation [20]. Developed at UC Berkeley, Ptolemy is a framework for modeling, simulation, and design of concurrent, real-time, embedded systems. The designs are intended to be *domain polymorphic* or in other words they can interact with each other in different domains. The model of computation is chosen by a 'director' which specifies the rules for the design. Thus Ptolemy is a software architecture design language.

SHIM: SHIM or Software/Hardware Integration Medium is an imperative approach to code synthesis with an asynchronous communication model [59]. It is a concurrent, asynchronous and deterministic model for simulation and synthesis targeting heterogeneous embedded systems. The idea is inspired by KPN model, but the buffers in KPN model were replaced by a rendezvous mode of communication.

Code synthesis frameworks are being developed to assist the engineer in programming complex designs and thereby reduce man hours, promote reuse and make verification easier. Formalisms behind some of the industrial tools like SIMULINK are not correct-by-construction and thus verification time is not reduced. Synchronous programming tools are correct-by-construction which give guarantees about generated code, if the higher level model is modeled correctly. Unfortunately, synchronous tools lag behind others like LabVIEW and Mathworks in creating user friendly GUI and tools to assist the engineer. We work on polychronous formalism where the availability of visual frameworks is even lower. But the main focus of code synthesis tools is in the code generation strategy rather than visual enhancements. The only available polychronous compiler Polychrony [61] constructs a clock tree of signals in the design and uses ROBDDS to form a clock hierarchy. We intend to replace this strategy with a prime implicate based strategy while remaining in the Boolean domain for analysis. Our visual formalism aims to be a more intuitive exposition for not only modeling but also for characterizing implementability of the specification as software programs.

Synthesis tools for hardware design and verification

As opposed to the limited progress in synthesis of embedded software, hardware design and verification has seen significant amount of automated code generation (Register Transistor Logic - RTL). Ahuja et al. [31, 23] present a methodology, which utilizes the power estimation knowledge to guide power reduction. The premise to their work is to first provide an accurate and efficient power estimation framework at higher abstraction level and then utilize this information to guide the power reduction algorithm while generating the hardware RTL. On the power estimation side, they present mainly three different approaches a-) reusing the RTL power estimation frameworks at higher level, b-) providing characterization based power models to facilitate the power estimation at high-level, c-) utilize various verification collaterals such as assertions and properties to speedup the design flow for power estimation.

The idea to reuse RTL power estimation frameworks in HLS based design methodologies comes from the fact that RTL is tool generated and the biggest bottleneck is the processing of simulation dump (vcd/fsdb). In [28], Ahuja et al. provide a rational for their approach and show that based on modeling style at system-level the speedup in estimation time can be achieved. They experimentally show on variety of benchmarks the range of speedup in power estimation process reaches upto 15times as compare to RTL power estimation techniques. The error or loss of accuracy on their proposed methodology was less than 10% with respect to lower level power estimates. Such techniques with more accuracy can be used for relative power estimations to detect trojans. More specifically work in [24], [25] present characterization based power estimation methodology, they utilize GEZEL based co-simulation environment. In this environment a hardware modeled as FSMD in GEZEL can be co-simulated with existing processor models such as ARM. The power model proposed in this case study is a regression based power model which is learnt through various statistical test data. The purpose to use a regression model is to utilize various states of fsmd as regression variables as opposed to input and output based characterization in previous approaches. This gives a better visibility on power estimation because this approach helps in relating activity of the states of FSMD to power consumption.

The estimates from the power estimation methodology/tools are no good if they are not supplied with good representative test vectors. Ahuja et al. in their work in [27], [26] present methodology to utilize verification collaterals to enhance the accuracy of power estimation at higher level. Similar case studies can help us in finding out trojans or relatively peak power test vectors. This work also shows how to write negative properties to help in creating counter cases to provide reachability of a state, transitions from a state and staying in particular state of the controller using invariants.

Ahuja et al. [30] present a power reduction based on clock-gating and sequential clock-gating from high level model description. To extend the clock-gating based power reduction Ahuja et al. [29] show how sequential clock-gating opportunities can be identified at higher abstraction level. Generally the difficulty to implement the sequential clock-gating comes from two facts 1-) if designer is trying to find such opportunities they might change the behavior of the design, 2-) If tools are used to facilitate such an optimization they generally work at netlist level try to find out the observability don't care or stability conditions at netlist level. This hinders the exploration of such an optimization at high-level. Clock gating and operand isolation are two techniques to

reduce the power consumption in state-of-the art hardware designs. Both approaches basically follow a two-step procedure: first, they statically analyze a hardware circuit to determine irrelevant computations. Second, all parts which are responsible for these computations are replaced by others that consume less power in the average case, either by gating clocks or by isolating operands. Jens et al. [49] defines the theoretical basis for adoption of these approaches in their entirety. They show how irrelevant computation can be eliminated using their approach. They present passiveness conditions for each signal x, which indicate that the value currently carried by x does not contribute to the final result of the system. After showing how their theory can be generally used in the context of clock gating and operand isolation a classification of many state-of-the-art approaches is performed and shown that most of the approaches in the literature are conservative approximations of their general setting. The author has worked on hardware design of redundant binary adders and multipliers which used many hardware generation tools from industry [89, 90, 91, 82].

3.5 Verification tools used in this work

Traditionally, software synthesis was followed by an orthogonal verification by testing, theorem proving, etc. Recently, verification tools were shown to be effective for use in code synthesis stage for generic computing algorithms [128]. Synthesis problems were encoded as verification problems, program guards were evaluated and program execution statements were inferred. The synthesis strategies we propose in the following chapters also utilize verification tools such as SMT solvers and model checkers for causal loop analysis. Here we provide information about the some of the tools we have used.

SAT based formal verification has been extensively used in equivalence checking, test generation and many other highly time consuming applications [68]. A SAT problem searches for a possible solution to a logical formula which is encoded in Conjunctive Normal Form (CNF). A CNF input is a conjunction of clauses where each of them is a disjunction of literals. If there exists a variable assignment which makes the formula true , then the problem is said to be satisfiable. Faster algorithms, better coverage, memory cost are some of the goals in this area of research. Since SAT solvers work with Boolean variables, a given specification has to be converted to Boolean equations where some of the information might be lost. Constraints which are data dependent may require the theory of Integers,theory of Reals, etc. This is a motivating reason to use Satisfiability Modulo Theories (SMT) for problems where relative values of variables affect the verification result. An SMT solver is a constraint solver which can decide if a formula encoded using non-Boolean variables is satisfiable [46, 54]. Recent advances in SMT solvers have made the use of these tools more popular in verification and synthesis domain [36, 46, 54]. We use the Yices SMT solver where formula is given as a set of assertions [21]. The satisfiability decisions made by Yices SMT solver is used to make our implementability decisions.

Model checking is another SAT-based formal verification tool that we have used in this research. Here, the model of a given system is built, and the satisfiability of a given formula is verified on

the model. Reachability analysis is one of the verification procedures made possible by model checkers. Here, the reachability of a particular state in question is analyzed, given a set of initial conditions. In our work, we use the Cadence SMV tool for reachability analysis [18]. Our work does not involve development of these verification tools. Our contribution is towards the application of these tools in research problems where we find their integration appropriate and fruitful.

Chapter 4

Analysis of Composition of Synchronous Systems

SIGNAL programs are synthesized into ANSI C code using the Polychrony compiler [61]. Polychrony compiler generates C code only for *endochronous* SIGNAL programs. Informally, a SIGNAL specification is said to be endochronous if a static sequential schedule exists for its operations. Currently Polychrony tool generates sequential C code from the concurrent SIGNAL specifications. A less strict property is *weak endochrony*. This property is satisfied even if a sequential schedule is not possible, but a locally concurrent schedule (with nondeterministic execution order for some operators) keeps the outputs deterministic. Thus if multiple possible behaviors without affecting the outputs are admissible for a specification, then such a SIGNAL specification is weakly endochronous. A more detailed account of endochrony, weak endochrony., etc with adequate examples can be found in Chapter 2. These properties are crucial in the design of GALS architectures for the ordering and distribution of operations into different sites.

In a GALS deployment of the code generated from various modules, due to unequal delay in communicating various signals from one module to another module and due to the absence of a global clock, the communication between processes will surely lose synchronization. Endochrony concerns with relations between rates of signals in a component and is not sufficient to guarantee correct communication. As discussed before, if a property such as *Isochrony* is satisfied for the common variables between the various synchronous modules one might be able to ensure correct communication [39] without losing the correctness of the system.

Isochrony is a condition that is applicable to a collection of synchronous modules which communicate through common variables. When the communication is synchronous (zero time), the time instant at which the updating module updates a shared variable, and the time instant when the modules reading the update on those variables see the update are the same. However, when the modules are distributed and their communication is asynchronous (GALS deployment), this is no longer true. The conditions on the updates and sampling of these shared variables must have some relationship in order for the GALS deployment to work the same way as the fully synchronous

model. [38, 39] defined *endo-isochronous systems* whose components are endochronous and the communication between them to be isochronous. There, isochrony has been studied based on the composition of synchronous transition systems (STS). A pair of STS (Φ_1, Φ_2) is said to be isochronous, if the asynchronous observations (traces) on the synchronous composition of the set of STSs and the asynchronous observations of asynchronous composition of desynchronized STS can be equated [38]. From [38],

$$(\Phi_1 \mid\mid \Phi_2)^a \;=\; \Phi_1^a \mid\mid_a \Phi_2^{a1} \tag{4.1}$$

This definition of isochrony was shown to hold when two STSs Φ_1 and Φ_2 have a set of common variables with the following property: If V be the set of variables shared between Φ_1 and Φ_2, and $V_1 \subseteq V$ is updated by Φ_1, and sampled by Φ_2, and $V_2 \subseteq V$ is updated by Φ_2 and sampled by Φ_1, and $V = V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$, then all variables in V_1 must have the same clock, and all variables in V_2 must have the same clock. In other words, if at any instant, one variable of V_i $(i \;=\; 1, 2)$ is absent (have no value), then all variables of that V_i must be absent. However, it turns out that this is only a sufficient condition, and not necessary. The actual definition of isochrony can be satisfied by more general conditions on the variable sets V_1 and V_2. There are several advantages of this generalization. First, it simplifies the understanding of isochrony substantially compared to [38, 39]. Second, it generalizes it to weakly endochronous modules also. Third, a directional notion of isochrony comes up naturally.

4.1 Preliminary definitions for synchronous systems

We use SIGNAL language to explain the composition of synchronous systems. An introduction for SIGNAL language was provided in Chapter 2. Here we define signals and their clock relations required to formalize properties like endochrony and isochrony.

Definition 4.1.1. *Given a set of signals V, each signal $x \;\in\; V$ has its own clock $\hat{x}$ which is the (possibly infinite) set of instants at which x gets updated. If the signal x is input to the system, $\hat{x}$ is the rate at which x is sampled. The sampling rate must be sufficient to capture all its updates.*

The clock of a signal specifies when the signal is valid. When two signals are used in a SIGNAL statement on either sides of the assignment operator, their clocks are said to be related. Now we formally define a clock relation.

Definition 4.1.2. *Given a set of signals V, two signals $(x, y) \;\in\; V$ is said to be related by clock relation r_{xy}, if their respective clocks $(\hat{x}, \hat{y})$ satisfies the relational predicate r_{xy}. For example, if the clocks of x and y are equal, then $r_{xy} \;=\; (\widehat{x} \;=\; \widehat{y})$.*
If we consider the definition of a signal (say x) in terms of two other signals (y, z), then they are related by a relational predicate r_{xyz} where $x, y, z \;\in\; V$ and x is defined in terms of y and z. For example, in the Merge statement $x \;:=\; y\; default\; z$, $r_{xyz} \;=\; (\widehat{x} \;=\; \widehat{y} \;\cup\; \widehat{z})$.

[1]Here Φ^a stands for desynchronized STS and $||_a$ means asynchronous composition

In the Table 6.1, the signals are equated in a direct fashion without the need for solving any external equation. This is termed as a *direct relation*. Now, two signals are said to be *indirectly related* if they are not present in a direct relation and if a relation can be found between them by solving a set of clock relations. Now we introduce a special clock relation '#' as the *null clock relation* which defines an absent clock relation between two signals.

Definition 4.1.3. *The signals x and y are independent if and only if the clock relation $r_{xy} = (\widehat{x} \,\#\widehat{y})$.*

This new relation can be used to assert that there exists no direct or indirect relation between the clocks of the signals under consideration. With these clock relations in place, we can now formalize the composition of different types of synchronous systems.

4.2 Composition of synchronous systems

In GALS systems, each component will need to interact in an asynchronous fashion over communication lines. The nature of the communication medium defines the type of composition required between components. In an asynchronous communication medium there is a requirement of buffers in between modules to balance the rates of individual modules. In practice, the size of the buffers cannot be infinite, which limits the difference in the rates of the modules. Another inhibiting factor for the proper functioning of the system is the information that is being sent between modules. The issues related to this factor are two fold: i) The clock of the signals which are common to the sender and receiver have to be shared and ii) The clock relationship between the common signals will have to be maintained in both modules. Here we consider the composition of endochronous and weakly endochronous processes. We assume that the programs or IPs provided for composition do not have combinational loops and are error free. Also when the programs or IPs are composed, they have to be checked for the presence of any new combinational cycles created as a result of the composition.

4.2.1 Composition of endochronous processes

A static schedule exists for an endochronous process. The different input signals have their own clocks and they are interconnected with the rate relations. A root clock will be formed as a result of these rate relations in the Polychrony compiler, which is the equivalent of a global clock in the SIGNAL domain. Thus, for an endochronous process, there exists a direct or indirect relation between every signal in the process. The advantage of such a process is that a flow of each signal (an order of values without clock) along with the clock relations is enough to create the clocked synchronous flow for the whole process, i.e. no external clock information is needed to recreate the synchronous flow of signals in an endochronous process. In SIGNAL terminology this is termed as

'reconstruction' of signals in a process [39]. Still endochrony is not sufficient to guarantee correct communication. The example below will illustrate this problem.

```
process Endo1= (? integer a,b;  ! integer out1,out2;)
 (| out1 := a + b | out2 := a + 1 when out1 > 0 |) end;
 process Endo2= (? integer out1,out2;  ! integer c;)
              (| c := out1 + out2|) end;
```

Here process `Endo1` has two output signals which are received by `Endo2` through an asynchronous medium. According to the order of arrival of inputs to `Endo2`, the value of c will change. Two cases of reconstruction of c for different rates of $out2$ is shown below.

a	1	0	2
b	-2	1	0
$out1$	-1	1	2
$out2$	$\perp$	1	3

$out1$	-1	1	2
$out2$	1	3	
c	0	4	

In `Endo1`, the values of $out2$ are updated only in the instants when $out1 > 0$. After transmission through an asynchronous medium, the synchronization between $out1$ and $out2$ are lost and therefore the absent values of $out2$ can not be observed for `Endo2`. `Endo2` can only observe a sequence of data of $out1$ and $out2$ respectively as shown on the right. By `c := out1 + out2`, `Endo2` assumes that $out1$ and $out2$ are updated at the same time and which contradict with the way they are being produced by `Endo1`. So endochrony is not sufficient for deterministic output while composing processes. Along with deterministic computation, deterministic communication has to be maintained. *Isochrony* is the property about the communication between systems which specifies whether they can be composed correctly. For endochronous processes we can guarantee proper composition provided certain conditions are satisfied. Before moving into the definitions which will check for isochrony in composing processes, we revisit some information that we know from existing literature in the form of assertions. From [39], we understand a strict condition for composing endochronous processes.

Assertion A1: For endochronous processes P_1 and P_2, isochrony can be guaranteed if P_1 produces all its events at the same instant and P_2 waits for all inputs to arrive before computation and vice versa.

This assertion from [39] is a sufficient condition for endochronous processes. In a GALS domain, this is often a strict condition and is hard to implement. The key factor in the composition process is the clock information of the composing IP's. So we put Assertion **A1** in a more general form for synchronous processes:

Assertion A2: For two communicating processes P_1 and P_2, the clock information of the common signals must be shared.

Sharing of clock information between IPs in a distributed environment involves exchanging meta-information about the clock relations for these endochronous processes. This information will

include the signals and their rate relations which can be used to check for isochrony. It is not possible to add any clock information into the IP, once it is implemented and shipped as an IP. So the meta-information will help in checking for isochorny and to redesign the IPs to make them isochronous. Now we formalize a sufficient condition for isochrony. For two endochronous processes P_i and P_j, where the set of signals of is V_i and V_j respectively, we define a communication structure C_{ij}.

Definition 4.2.1. *The tuple $C_{ij} = (W_{ij},\ R_{ij})$ defines a communication structure which represents the projection of clock signals from P_i to P_j, where*
i) $W_{ij} = V_i \cap V_j$ *and*
ii) $R_{ij} = \{\ r^{ij}_{abc}$ *, the clock relation defining* $\widehat{v_a}$ *in terms of* $\widehat{v_b}$ *and* $\widehat{v_c}$ *in* P_i *,* $\forall\ (v_a, v_b, v_c) \in W_{ij} \times W_{ij} \times W_{ij}\}$

The communication structure consists of the shared signals in both processes and the clock relations between them. The relation r will depend on the relation between signals in P_i. For an endochronous process, there exists a direct or indirect relation between any two signals. Now **A2** can be rewritten as follows.

Assertion A3: Given two processes P_i and P_j with communication structures $C_{ij} = \{\ W_{ij}\ ,\ R_{ij}\}$ and $C_{ji} = \{\ W_{ji},\ R_{ji}\}$ respectively, if $R_{ij} = R_{ji}$, then P_i and P_j are isochronous.

The common signals and their rate relations can be checked for a match using the meta-information that is extracted from these IPs. The IPs themselves do not have the knowledge of the similarity in rate relations, albeit they are behaving in isochrony. Now, let's analyze `Endo1` and `Endo2` again based on assertion A3. Since `Endo1` and `Endo2` communicate via signals $out1$ and $out2$, therefore $W_{12} = W_{21} = \{out1, out2\}$. Next, let's observe the clock relation between $out1$ and $out2$. In `Endo1`, $R_{12} = \{r^{12}_{out1out2} = (\widehat{out2} = \widehat{a} \cap [\widehat{out1 > 0}])\}$, in `Endo2`, $R_{21} = \{r^{21}_{out2out1} = (\widehat{out2} = \widehat{out1})\}$, therefore $R_{12} \neq R_{21}$ and doesn't meet assertion A3.

This example fails to meet isochrony requirements in two ways. Firstly, it does not define relation between $out1$ and $out2$ in the same manner in both IPs. Secondly, the output signals are not defined in terms of shared signals in the first IP. From Definition 4.2.1, we can see that the relations are between signals present in the set of shared signals W_{ij}. So the communication structure has to be complete with the signals which define relations between shared signals. For understanding how Assertion **A3** validates isochrony property for two endochronous processes, we modify the example discussed above. The endochronous process `Endo1` is composed with a new endochronous process `BEndo2`. The rate relations for `BEndo2` are as follows: $R_{21} = \{r^{21}_{out2out1} = (\widehat{out2} = [\widehat{out1 > 0}])\}$. Here the rate relations for the communicating signals are maintained ($R_{12} = R_{21}$) in both processes and thus isochrony is verified. Another way of correcting the example would be to share the signal a along with $out1$ and $out2$ and maintain the relation between them in the communicating IP. Since correcting IP blocks while composition is not feasible, care has to be taken to ensure that the communication structure and relations are *complete*, during the design process of an IP.

```
process Endo1= (? integer a,b;  ! integer out1,out2;)
   (| out1 := a + b | out2 := 1 when out1 > 0 |) end;
process BEndo2= (? integer out1,out2;  ! integer c,d;)
     (| c := 1  when out1 > 0| d := c + out2 |) end;
```

4.2.2 Composition of weakly endochronous processes

A process is said to be weakly endochronous if it satisfies the *diamond property* defined in [119]. Weak endochrony is a less strict form of endochrony where there are unrelated signals which can be scheduled in multiple orders, but still generate a deterministic output. This eases the clock constraints and makes composition of processes simpler. An example of two weakly endochronous process is given below:

```
process WEndo3= (? integer a1,c1;  ! integer b1,b2;)
    (|   b1 := a1 + 1 | b2 := c1$ init 0 + 1 |) end;
process WEndo4= (? integer b1,b2;  ! integer c1,c2;)
       (|   c2 := b1 + 1 | c1 := b2 + 1 |) end;
```

Here there is bidirectional communication between processes. The communication structure C_{34} and C_{43} will be equal for common signals. W_{34} and W_{43} will contain `b1,b2,c1`. We know already from [119] that the two processes `WEndo3` and `WEndo4` will have correct composition and are isochronous. Assertion **A3** can be verified by evaluating the clock relation set $R_{34} = \{(\widehat{b2} = \widehat{c1})\}$ and $R_{43} = \{(\widehat{b2} = \widehat{c1})\}$. Another important factor is the independence of clocks of signals in weakly endochronous processes. In `WEndo3`, the clocks $(\widehat{b1}, \widehat{b2})$ and $(\widehat{b1}, \widehat{c1})$ are totally independent. Similar is the case in R_{43}. To formally express this situation, we use the *null clock relation* to define extended clock relation sets R'_{ij}, R'_{ji} and their extended communication structure C'_{ij}, C'_{ji} for two weakly endochronous P_i, P_j.

Definition 4.2.2. *For weakly endochronous processes P_i and P_j, their extended clock relation sets are $R'_{ij} = R_{ij} \cup \{\#\}$, $R'_{ji} = R_{ji} \cup \{\#\}$ and their extended communication structures are $C'_{ij} = \{W_{ij}, R'_{ij}\}$, $C'_{ji} = \{W_{ji}, R'_{ji}\}$.*

This definition denotes that unlike endochronous processes, for two weakly endochronous processes P_l and P_m with clock relations $r \in R'_l$ and $r \in R'_m$ can be equal to '#', or say null clock relation. It has to be noted that null clock relation means that there should not be any relation between signals rather than the relation is unspecified. We will use this extended clock relation set to evaluate isochrony for two weakly endochronous processes.

Assertion A4: Given two weakly endochronous processes P_i and P_j with communication structures $C'_{ij} = \{ W_{ij},\ R'_{ij}\}$ and $C'_{ji} = \{ W_{ji},\ R'_{ji}\}$ respectively, if $R'_{ij} = R'_{ji}$, then P_i and P_j are isochronous.

We analyze Assertion **A4** by analyzing the extended clock relation set R'_{34} and R'_{43} of `WEndo3` and `WEndo4`. The shared variables between `WEndo3` and `WEndo4` are b_1, b_2, and c_2. Therefore, we have $R'_{34} = \{r^{34}_{b1b2} = (\widehat{b1}\#\widehat{b2}), r^{34}_{b1c1} = (\widehat{b1}\#\widehat{c1}), r^{34}_{b2c1} = (\widehat{b2} = \widehat{c1})\}$ and $R'_{43} = \{r^{43}_{b1b2} = (\widehat{b1}\#\widehat{b2}), r^{43}_{b1c1} = (\widehat{b1}\#\widehat{c1}), r^{43}_{b2c1} = (\widehat{b2} = \widehat{c1})\}$ thus $R'_{34} = R'_{43}$. Based on Assertion **A4**, P_i and P_j are isochronous which confirms with the known result. Since any endochronous processes is also a weakly endochronous processes, Assertion **A4** also works for two endochronous processes. This demonstrates a uniform way to sufficiently evaluate correct communication (isochrony) for two endochronous or two weakly endochronous processes.

4.3 Directional Isochrony

In the previous sections, we have discussed the composition between two endochronous processes or two weakly endochronous processes. The composition of an endochronous process to a weakly endochronous process is of special interest to GALS domain. Adding new modules to existing designs without bidirectional communication can be verified easily if the assertions for this case hold true. Before going into the definitions related to the composition, let's observe two examples to understand the significance of the direction of communication.

```
process Endo5= (? integer a1,b1;  ! integer c1,d1;)
(| a1 ^= when b1 > 0 | c1 := a1 + 1 | d1 := b1 |) end;
process WEndo6= (? integer c1,d1;  ! integer e1,f1;)
        (| e1 := c1 + 1 | f1 := d1 + 1 |) end;


process WEndo7= (? integer a2,b2;  ! integer c2,d2;)
        (| c2 := a2 + 1 | d2 := b2 + 1 |) end;
 process Endo8= (? integer c2,d2;  ! integer e2,f2;)
 (| c2 ^= when d2 > 0 | e2 := c2 + 1 | f2 := d2 |) end;
```

One can easily find that `Endo5` is an endochronous process and is similar to `Endo8` and `WEndo6` weakly endochronous process which is similar to `WEndo7`. The two compositions differ only by their communication direction. In the composition of `Endo5` and `WEndo6`, the extended communication structures of `Endo5` and `WEndo6` are $C'_{56} = \{W_{56}, R'_{56}\}$ and $C'_{65} = \{W_{65}, R'_{65}\}$, where $W_{56} = W_{65} = \{c1, d1\}$, $R'_{56} = \{r^{56}_{c1d1} = (\widehat{c1} = \widehat{[d1 > 0]})\}$, $R'_{65} = \{r^{65}_{c1d1} = (\widehat{c1}\#\widehat{d1})\}$. Therefore, R'_{56} is not equal to R'_{65}. We analyze the second composition between `WEndo7` and `Endo8` in the same manner and we have $R'_{78} = \{r^{78}_{c2d2} = (\widehat{c2}\#\widehat{d2}\}$ which is not equal to $R'_{87} = \{r^{87}_{c2d2} = (\widehat{c2} = \widehat{[d2 > 0]})\}$.

The composition of `Endo5` with `WEndo6` is correct since the statements in `WEndo6` are concurrent. The clock dependencies in `Endo5` are not relevant, since the statements in `WEndo6` are

independent. `WEndo6` can be reconstructed with multiple behaviors which will have the same output. On the other hand, the communication from `WEndo7` to `Endo8` results in loss of values since `Endo8` get values of $c2$ only when $d2 > 0$ holds. From these examples we can conclude:
1) Assertion **A4** is not a necessary condition for isochrony since it may reject the case like composition of `Endo5` and `WEndo6`;
2) Correct communication between an endochronous process and a weakly endochronous processe is determined by its communication direction;
Therefore, an endochronous process can be composed to a weakly endochronous process, if the direction of communication is unidirectional.

Definition 4.3.1. *Two processes P_i and P_j are directionally isochronous $P_i \overrightarrow{||} P_j$, if and only if the correctness of communication from P_i to P_j is guaranteed while the opposite direction is not.*

By correctness of communication, we mean the flow equivalence of the communicating signals. The order is preserved after asynchronous communication and the signals are not reconstructed in a manner, which contradicts the rate relations in the sending process. To test for directional isochrony, the null clock relation is used along with the normal set of relations.

Definition 4.3.2. *A partial order '$<$' between a determined clock relation r and null clock relation '$\#$' is defined as $\# < r$.*

This partial order will lead to a partial order $\preceq$ between any two extended clock relation sets R'_{ij} and R'_{ji}.

Definition 4.3.3. *For two composed processes P_i and P_j, $R'_{ij} \preceq R'_{ji}$ if and only if $\forall r_{ab}^{ij} \in R'_{ij}$ and its corresponding $r_{ab}^{ji} \in R'_{ji}$, there exists a partial order such that $r_{ab}^{ij} \leq r_{ab}^{ji}$.*

Assertion A5: Given one endochronous P_i and one weakly endochronous processes P_j, $P_i \overrightarrow{||} P_j$ if and only if $R'_j \preceq R'_i$.
For our examples shown at the beginning of this section, $R'_6 \preceq R'_5$ and therefore `Endo5` $\overrightarrow{||}$ `WEndo6`. The communication from `Endo5` to `WEndo6` is correct while the opposite direction cannot be guaranteed. We can observe similar behaviour for `WEndo7` and `Endo8`. Assertion **A5** is not limited for only (endochronous, weakly endochronous) processes, it can also be used to analyze any two weakly endochronous processes by checking the clock relations between any pair of shared variables.

Related work based on this property has attempted to model asynchronous implementations of synchronous specifications [118]. There it is shown that the asynchronous implementation is a correct refinement of its synchronous specification and hence the GALS implementation can be performed from synchronous models. Distributed implementation of synchronous programs require extendability of the design for improving or adding a new IP to the existing system. By extendability, we mean the composition and distributed implementation of a new IP into the whole system. The endochronous process currently generate sequential code which can be extended only if the whole

system is recompiled and the assertions **A1**, **A2** and **A3** are satisfied. Weakly endochronous programs are more suited for extension as well as for distributed implementation due to its concurrent nature and null clock relations. This applies to the weakly endochronous processes in a directionally isochronous system. By providing the formalization for checking isochrony based on the SIGNAL synchronous environment, we have provided a method to verify the composition of synchronous IPs to be used in a GALS environment.

Chapter 5

Code generation from synchronous data flow models

Synchronous programming languages and their compilers have been developed with the goal of generating lower level code (C, HDL, etc.) while preserving the deterministic nature of outputs of systems modeled using them. Properties have been defined which ensure deterministic operation and conformance to these properties is a necessary condition for code synthesis. While the correct-by-construction code synthesis has made generation of correct sequential code possible, multi-threaded code generation using synchronous programming tools is still at best an experimental feature in the tool set. Due to our long association with the von Neumann sequential programming models, it is often hard to write correct multi-threaded code [103]. There has been several initiatives such as defining new programming languages, adding pragmas or directives to existing languages and so on to help a programmer to express specification in parallel. In synchronous programming languages, since concurrency of program statements is a given, the challenge is to characterize polychronous specifications which are implementable as multi-threaded C code. There is also a need to explore if the multiple behaviors possible while executing a multi-threaded code, can be mapped back to easing of restrictions for code generation.

For research experimentations on multi-threaded code generation, we choose the synchronous programming language SIGNAL [71] and its associated Polychrony compiler [61]. Our objective is to find how to infer the threading structure by analyzing the SIGNAL specifications through its clock relations and data flow analysis. We also bring out the fact that the SIGNAL program does not have to be endochronous to be multi-threaded, and weak-endochronous SIGNAL programs can also be multi-threaded. Also we investigate a *noninvasive* methodology for utilizing polychronous data flow specifications using an existing compiler for SIGNAL. In doing our investigations, we hope to provide a means for extending the SIGNAL compiler with better granularity in its thread-based code generation.

5.1 Endochrony, Weak Endochrony, and Multi-Threading

Let us consider an example from [114] of a pair of SIGNAL processes, Producer and Consumer, which themselves are endochronous, but their composition is not endochronous. However, since endochrony implies weak endochrony (Since endochrony allows static schedule, the diamond property trivially holds), and weak endochrony being compositional [119], their composition is weakly endochronous. We illustrate through this example, that one can generate correct multi-threaded code from weak-endochronous processes, and hence the thread generation process is compositional. One can generate threads for each weak endochronous process, and then while putting them together, can synthesize the basic synchronization between threads across the processes, and obtain a correct multi-threaded program.

```
process Producer =
   (? boolean a; ! integer u, x;)
   (|u := (1 + (u$1 init 0)) when a
    |x := (1 + (x$1 init 0)) when not(a)|)
    end;
process Consumer =
   (? boolean b; integer x;
    ! integer v;)
   (|v := (v$1 + (x default 1))
    |x ^= when b
    |v ^= b  |)
    end;
```

The producer process has one input and two output signals, and the computations on the output signals are under mutually exclusive conditions, and hence easily seen to be endochronous. The process Consumer, however, has two clock equations, stating that its output v is synchronous with input b. But the value of v uses a value from signal x only when b has a `true` input. Due to these clock equations, this process is also endochronous. However, a composition of these two, when they share signal x produced by the producer, and consumed by the consumer v is not isochronous due to the fact that a, b input signals for the composed process does not have any implied relationship. Only fact that can be derived is; whenever a occurs with a `false` value, a new value on signal x is produced. Now for that value on x to be read by the consumer, its triggering condition (b is `true`) must occur at the same time. However, when a is `true`, then b may or may not occur. If it does occur, it must have a `false` value, otherwise the consumer will expect a new value on signal x.

The inferred facts from the SIGNAL program is depicted in Figure 5.1, which shows that the composed process is weakly endochronous. In Figure 5.1 it is shown that the result will be same as in the synchronous case if a has a `true` occurrence and b has a late `false` occurrence or when the order of occurrence is reversed. The other possibility is a occurring as `false`, and b as `true` as demanded by the specification. It is not possible that a occurs with `false` and b also occurs with `false` or both occur as `true`. The implications of the specification disallow that. Figure 5.2 shows the corresponding multi-threading structure. In the generated multi-threaded C code, therefore, a is read by one reader thread R1, and b is read by another reader thread R2. If a

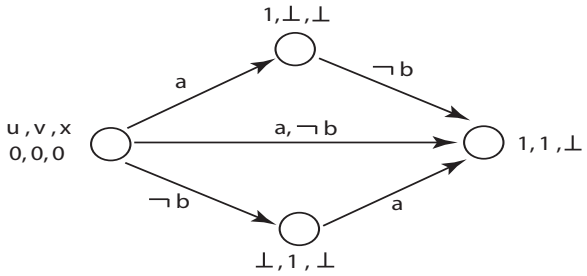


Figure 5.1: The Weak-Endochrony or Diamond Property of Producer/Consumer Composition

occurs with `true` then R1 notifies the thread P1 that computes u and new value of u is generated. If b occurs with `false`, then R2 notifies the thread C2 that increments v by 1. However, if a occurs with `false`, then R1 notifies the thread P1 that generates new value of x. At the same time R1 notifies the thread C1 that computes the alternate value of v using x's new value. Since it is guaranteed by the constraint on the environment that a's `false` occurrence will be accompanied by a `true` occurrence of b, events $e2, e3$ occur together and C1 will be notified to update.

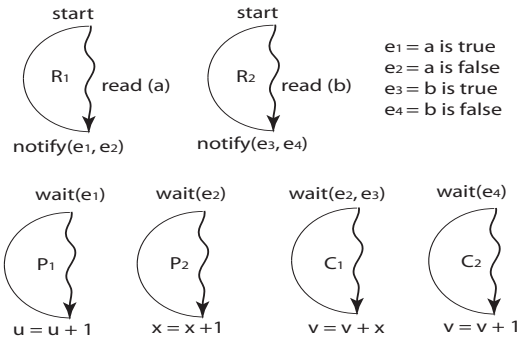


Figure 5.2: The Multi-Threaded Structure of the Producer/Consumer Composition

Here weak-endochronous programs are implemented as multi-threaded programs while preserving the semantics of the original SIGNAL specification. This also necessitates further research into how to easily check the weak endochrony property of SIGNAL programs. In the next section, we show how the internal formal data structure defined by [105] can be used for inferring the threading structure while generating multi-threaded code from SIGNAL specifications.

5.2 Synchronous Flow Dependence Graphs for Determining Threading Structures

A Synchronous-Flow Dependence Graph **(SFDG)** is a formal structure representing the data flow, clock relations, dependencies and scheduling relations inherent in a SIGNAL specification [105]. The nodes of the SFDG would represent the signal names in the code, while the arcs will represent the dependencies between the signals. A signal z is associated with its clock represented by $\hat{z}$. The clock of a signal defines the presence or absence of values. Any signal which has a data dependency on another signal is automatically related in terms of their clock logic. A simple code in SIGNAL is given below:

$$p := z + 1 \mid r := p \ when \ q$$

The signal p depends on z and since every new value computed on p requires a new value on z, hence $\hat{p} = \hat{z}$. The signal r is dependent on p and q. Every time r is computed, a value on p as well as on q are needed. A diagram depicting the clock dependencies of the three signals is shown in Figure 5.3(a). The clock relationship of these signals can be expressed as follows:

$$\hat{p} = \hat{z} \mid \hat{r} = \hat{p} \wedge \hat{q} \quad \text{or} \quad \hat{r} = \hat{z} \wedge \hat{q}$$

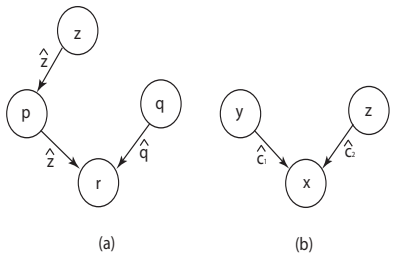


Figure 5.3: Clock relationship

One of the most repeated keywords in SIGNAL would be *default*. It is used to prioritize the update operation on a signal. An example for explicitly specifying a deterministic order is as follows:

$$x := (y \ when \ c_1) \ default \ (z \ when \ c_2)$$

Here the signal x is updated with y or z according to the boolean conditional signals c_1 and c_2. These signals c_1 and c_2 must be present and should hold true for the respective condition to be satisfied. The first condition c_1 is given a higher priority in the statement by the *default* keyword. The signal z can update x only when the condition c_1 fails and the condition c_2 succeeds. The clock dependencies associated with the *default* keyword is shown in Figure 5.3(b). The clock

relationship of the signals associated with the *default* keyword is as follows:

$$\hat{x} = (\hat{y} \wedge \hat{c_1}) \vee (\hat{z} \wedge \hat{c_2})$$

A code generation tool like Polychrony converts the SIGNAL code into C code. The clock logic associated with each signal is taken into consideration while the code is being translated. In a sequential execution environment, the *default* operator in SIGNAL can be easily represented by *if .. else* statements in C . In a multi-threading environment, if the conditional signals are present in different threads, the resolution of this statement will become difficult. If y and z are present in different threads, the order of updating x will be dependent on which thread is given the handle by the kernel. In order to ensure correctness of the code, a conditional *wait* and *signal* arrangement has to be performed in the competing threads. This arrangement will make sure that both the threads are called in the right order and at least once before the next set of inputs are being fed. Multiple threads have to arrive at a synchronization point, transfer values, receive new inputs and then proceed. In fact, unless sufficient constraints on $\hat{c_1}$ and $\hat{c_2}$ can be established, the default example above is not even weakly endochronous, and hence threaded version will be non-deterministic, which is not desirable. For example, if it can be established that $\hat{c_1} \wedge \hat{c_2} = \emptyset$ as is the case with Consumer process in the previous section, then two separate threads can update x without having a race condition. However, if $\hat{c_1} \wedge \hat{c_2} \neq \emptyset$, then we have to establish a synchronization mechanism implied by $\hat{c_1} \wedge \hat{c_2}$ that allows a deterministic assignment to x at the instants $\hat{c_1} \wedge \hat{c_2}$. If not, such a specification should be rejected by a code synthesizer. Given that we know how to construct the SFDG compositionally from the individual SIGNAL equations [105], the SIGNAL compiler does construct just that. Also, note that if there is a cycle in the graph $x_1 \xrightarrow{\hat{c_1}} x_2 \xrightarrow{\hat{c_2}} x_3 .... \xrightarrow{\hat{c_{n-1}}} x_n \xrightarrow{\hat{c_n}} x_1$ then $\bigwedge_{i=1}^{n} c_i = \emptyset$ must hold[105]. This is not implementable due to cyclic dependency which cannot be satisfied by any code, and hence will be rejected.

5.2.1 Inferring Threading Structure

Now that we know how to construct the SFDGs, and analyze the clock relations through the dependency structures, in this subsection we show how to use such analysis to infer threading structures for generating C code for a weak-endochronous process. Here we also show two examples for a better understanding of our method.

Inferring Pipelining Structure

Consider the following SIGNAL process P which consists of 3 subprocesses fmain, gmain, hmain. In Figure 5.4 we present the SFDG for this process with the clock constraints on the edges.

```
process P= (? integer value;
 ! integer f_out, g_out, h_out;)
```

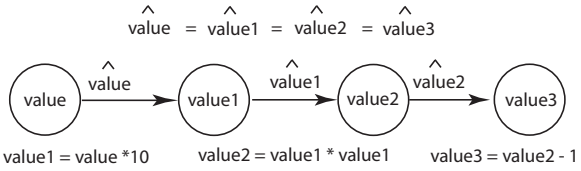


Figure 5.4: SFDG for Process P

```
(| value1 := fmain(value)
 | f_out  := value1
 | value2 := gmain(value1)
 | g_out  := value2
 | value3 := hmain(value2)
 | h_out  := value3 |)
where integer value1, value2, value3;

process fmain = (? integer fin;
 ! integer fout; )
 (| fout := fin *10  |);
process gmain =
  (? integer gin;
   ! integer gout;   )
  (| gout := gin * gin |);
process hmain =
  (? integer hin;
   ! integer hout;   )
  (| hout := hin - 1 |);
 end;
```

This SFDG is obtained by superimposing the SFDGs for each of the subprocesses and by renaming the signals onto the main process P. As can be seen, the linear structure of the SFDG in Figure 5.4 with the clocks on each edge being equal because $\hat{value} = \hat{value}_1 = \hat{value}_2 = \hat{value}_3$, this process can be synthesized into a pipeline threading structure. In the resulting sequential code, a new input at $value$ will need to be processed by three processes in a cascade fashion. If pipelined using a multi-threaded structure with shared state between fmain and gmain, and also one between gmain and hmain, the data rate can be increased by three times of the rate of sequential code. The buffered pipeline structure is shown in Figure 5.5. Pthread library's *conditioned wait and notify* constructs can be used to synchronize the write and read operations on buffers by each thread. All these can be automatically inferred. If the throughput demand is higher, one could also compute the buffer sizes for the buffers shared between the subsequent threads easily in this case. Note that, if an SFDG is linear, as is the case here, and the clock constraints on each edge are not equal but are related, adequate buffer sizes can be inferred by knowing the relationships between the clocks, or by guarding the buffers based on the correct *condition* structures of Pthreads library.

Inferring Constraints on the Environmental Inputs while Synthesizing

Now we show another version of the Producer/Consumer problem with a single place buffer between them. This discussion can be generalized to any finite size buffer. In this example, the

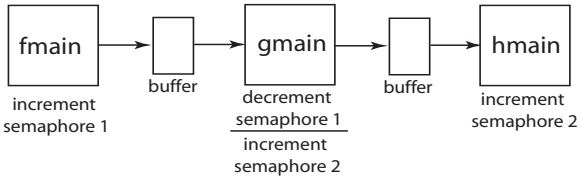


Figure 5.5: Pipeline model for synchronization

threading structure along the subprocess boundaries of the SIGNAL specification can be synthesized with the required synchronization mechanism based on *counting semaphores*. However, we do not show the inference of the synchronization primitive in this example. We rather explain how a synthesizer can also automatically infer the constraints on the environmental inputs that must be guaranteed to make the multi-threaded program work correctly. Such issues are often not clearly understood from the C code, or the C code is often over designed unknowingly. For example, a programmer might make a reader thread read the Boolean inputs p and c in the same loop. That would effectively imply that their clocks are of same rate ($\hat{p} = \hat{c}$) which may not necessarily be satisfied by an embedded environment, resulting in loss of stimuli on one of the inputs. Assume/Guarantee style of programming thus cannot be supported without formal analysis as proposed here. The following SIGNAL specification has a $producer$ process that takes as input a stimuli p, which is to be interpreted as requests to produce new data. In this specification, the data produced is the output of a counter modulo 7, but that is just a place holder for any computation, or it could be an output from a sensor. The $consumer$ process reads stimuli on another channel c and interpret it as a request from the environment to get data from the producer/consumer system. The Consumer can also read the value stored in the one place buffer b through channel $bvalue$, and then outputs on channel $pcvalid$ if the buffer contains new data or stale data. When $pcvalid$ is true , $pcdata$ will give out the output data.

```
process pcmain =
  (? boolean p, c;
   ! integer pcdata; boolean pcvalid;)
 (| pp ^= b ^= p ^+ c
  | pcdata ^= pcvalid ^= c
  | d := producer(p)
  | b := d cell c
  | pp:= (p$1 init false) default false
  | (pcdata,pcvalid) := consumer(c,pp,b)
  |)
  where boolean pp;
        integer d, b;

process producer =
  (? boolean ptick; ! integer dvalue;)
  (| counter := ((prev_count + 1)
            modulo 7 ) when ptick
   | prev_count := counter$1 init 0
   | dvalue := counter
   |)
   where
    integer counter, prev_count;
   end;
process  consumer =
```

```
  (? boolean ctick, u; integer bvalue;
   ! integer dread; boolean v;)
 (| v ^= dread ^=  ctick
  | v := true when u default false
  | dread := bvalue when v default -1
  |);
end;
```

In Figure 5.6 the SFDG of this SIGNAL specification is shown. One can see that the node for signal pp has conflicting assignment due to the rate constraint in the program ($pp \hat{=} b \hat{=} p \hat{+} c$), when $\hat{p} \wedge \hat{c} \neq \emptyset$, and $\hat{p} \neq \hat{c}$. There will be a race for changing the value of pp whenever $\hat{p} \wedge \hat{c} \neq \emptyset$. Therefore, we have a non-weakly endochronous process, which cannot be safely multi-threaded. Race condition can be avoided only if the constraints $\hat{p} \leq \hat{c}$, or $\hat{p} \wedge \hat{c} = \emptyset$ hold. If there is no guarantee for this constraint, then the generated multi-threaded program will have race conditions leading to non-deterministic behavior.

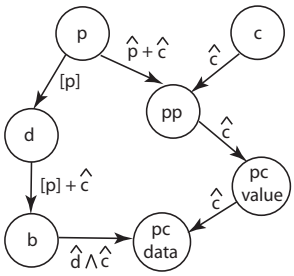


Figure 5.6: SFDG for Producer/Consumer Example

This example shows that not only our synthesis methodology can generate correct multi-threaded code from weakly endochronous SIGNAL specifications, it also may be able to warn the user, when weak endochrony is violated by the SIGNAL specification, with the set of constraints on the inputs that must be maintained to get safe and deterministic behavior from the resulting code.

5.3 Algorithm for constructing SDFG graphs to enable multi-threaded code generation

A given SIGNAL program P is a concurrent composition $P_1 \mid P_2 \mid ... \mid P_k$. If we flatten the process structure, then the entire program is actually concurrent composition of concurrent signal relations such as $x := x\$ \; init \; 0$, or $x := ((y\$ \; init \; 1) + 1 \; when \; z$ etc. The first example is what we call a normalized statement (NS) in SIGNAL, in the sense that the normalized expressions (NE) on the right hand side of the assignment is built from variables only. The second signal statement is what we call a complex statement (CS) because it contains on the right hand side of the assignment a complex expression (CE). Here, $((y\$ \; init \; 1) + 1) \; when \; z$ contains a normalized expression $((y\$ \; init \; 1)$ inside it. Given a SIGNAL program P, one can convert it into an equivalent normalized program P_N by converting the CS into multiple NS by adding additional SIGNAL variable names.

For example $x := ((y\$\ init\ 1)\ +\ 1)\ when\ z$ is equivalent to $x := x_2\ when\ z\ \mid x_1 := (y\$\ init\ 1)\ \mid x_2 :=\ x_1\ +\ 1$. Here by adding two intermediate additional variables x_1 and x_2, we have created an equivalent concurrent composition of normalized statements for the initial complex statement. This conversion is a crucial step in our algorithm. In this document we outline algorithmic steps needed to convert a SIGNAL program with complex statements into an equivalent SIGNAL program with only normalized statements. More details on the grammar and algorithms is provided in [96]. For multi-threaded code generation from SIGNAL programs, we follow these steps.

1. Given the signal program P, convert the program into an equivalent normalized program P_N. We number the statements as $1, 2, ...$ in the normalized program.
2. Given a normalized program P_N, construct a signal dependence graph G_P which has one node for each normalized statement. The node has multiple input ports corresponding to the SIGNAL names on the righthand side of the normalized statement, and one output port for the SIGNAL name on the left hand side of the assignment statement.
3. The links are annotated with a clock equation of the form $\hat{x} = \hat{y}$, if the link is between a node whose output port corresponds to signal x and a node whose input port for this link corresponds to a signal y.
4. Given graph G_P, we analyze the graph for weak endochrony (which is decidable [119]) using the clock relations , and any clock equations implied by the statements in P_N. For example, if P_N has a statement $x\ :=\ y\ default\ z$, then we have an implied clock relation $\hat{x} = \hat{y}\ \cup\ \hat{z}$.
5. We have proven [98], if the clock relations are proven consistent, and if the entire G_P with the clock relations are weakly endochronous, then if we have thread implementation of each of the node, the entire system of threads will be working correctly and will be flow equivalent to P. So this will be the last step of the code synthesis.

However, note that this code generation is non-optimal in the sense that it will create one or more threads for each normalized statement, which will result in threads with short execution times. This micro-threading scheme is not an efficient implementation as thread to thread communication will incur synchronization cost. Therefore coalescing of the threads based on their dependencies will be required for faster execution and different multi-threading strategies will have to be studied to lessen the synchronization overhead.

5.4 Multi-threaded code generation schemes for SIGNAL compiler

A recent enhancement of the SIGNAL compiler Polychrony generates multi-threaded code with tight synchronization between atomically executing threads. This is a valuable step towards man-

aging concurrency, but multi-threading if not judiciously employed has severe performance drawbacks due to the usage of too many threads resulting in high context switching overheads. In our opinion, the number of threads generated by the micro-threading scheme of the current SIGNAL compiler is larger than what is required. We illustrate this with the following pseudo SIGNAL code example:

```
process P = (? integer x; !integer y,z;)
(| y := x + 1 | z := x-1|)
```

This simple SIGNAL code fragment provides an input argument x and two output arguments y and z. The behavior of P specifies parallel execution of computing and assigning $x+1$ and $x-1$ to y and z respectively. Figure 5.7 shows the SIGNAL implementation and its thread-call structure.

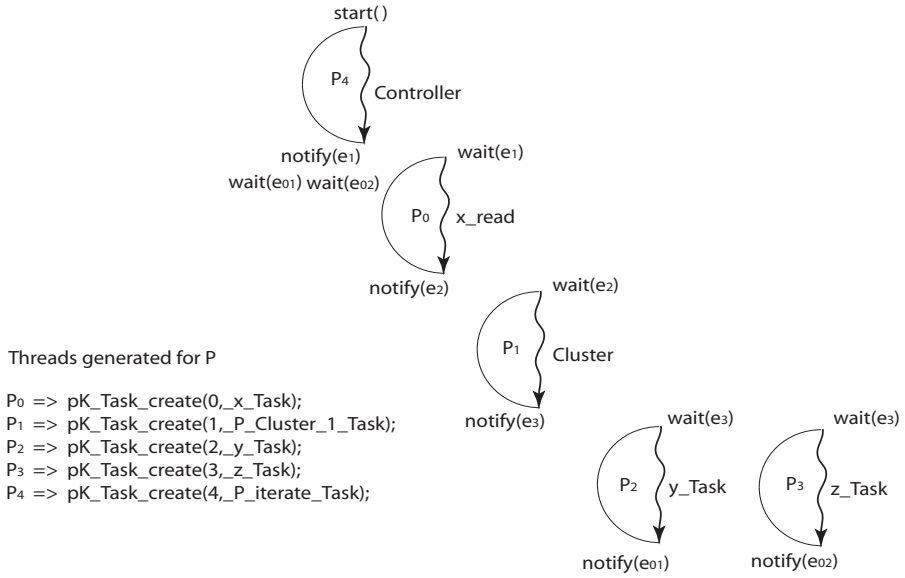


Figure 5.7: Current Micro-threading Structure by SIGNAL compiler

For this simple code fragment, the current SIGNAL compiler generates 5 threads; one for reading values of x, two for outputting y,z, one for the computation of the values of y,z based on new values of x, and one controller thread that sequentializes the reading of x, followed by the computation, and finally followed by writing y,z. The use of 5 semaphores to synchronize the sequentialization is very inefficient. In case the main computation is multi-rate, this compiler will generate several threads by clustering the computations driven by the same clock in single threads, and then again controlling the sequence using semaphores. We feel that such micro-level threading may be counter productive in terms of efficiency. Another problem is that the SIGNAL compiler is not open-source yet. So, if one does not agree with such micro-threading strategy in their current threading implementation, any invasive change in the compiler is not possible for external users.

Moreover, parallel architectures for embedded computing have a limited number of resources for multi-threading, which makes judicious selection of threads for parallelism extremely important.

Here we investigate a *noninvasive* methodology for utilizing polychronous dataflow specifications using the only available compiler for such specification. We make no internal changes in the compiler, but still generate multi-threaded code with wrappers. We can generate multi-threaded C code with deterministic even when the endo-isochrony is not satisfied between the partitioned parts of the data flow computation specified. The intention here is to show how multi-threaded C code can be generated using polychronous specifications and corresponding compilers for sequential code generation with designer imposed threading granularity and designer specified threading boundaries. This alleviates the problem of (i) having to write multi-threaded code by hand from a sequential specification of the computation being implemented; and (ii) micro-level thread granularity causing unnecessary performance overhead. We illustrate our methodology with a running example from the STARMAC project [22].

5.4.1 The STARMAC Example

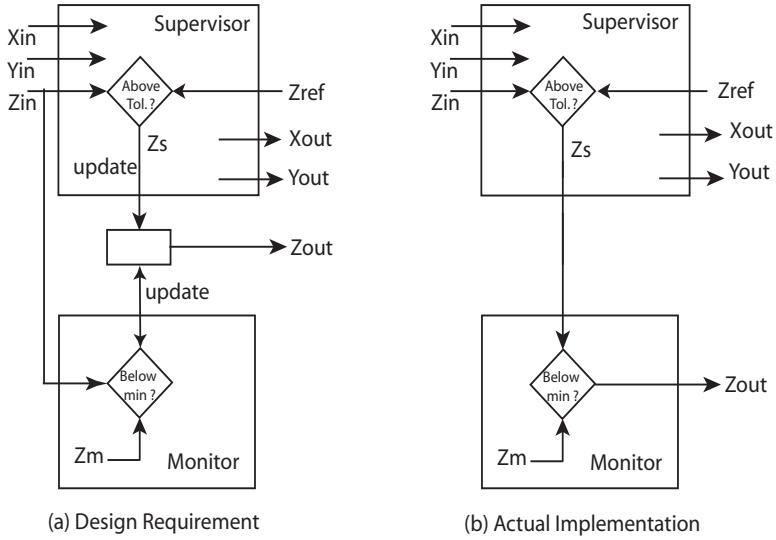


Figure 5.8: STARMAC Design requirement

A group at Stanford [22] is currently working on a testbed for multi-agent control called STARMAC. It contains four rotors that are controlled by a supervisory control unit managing the flight control commands. We present the height supervisory control unit in Figure 5.8 with its design requirement and eventual implementation using Simulink. Looking first at the design requirements in Figure 5.8(a), we see the `Supervisor` and `Monitor` entities. The `Supervisor` is responsible for reacting to the major functions of flight control such as computing possible next

coordinates to move to. The `Monitor` on the other hand only focuses on maintaining the elevation of the STARMAC testbed by proposing adjustments to the elevation when below a certain threshold. Labels X, Y, and Z represent the x, y and z coordinates of the STARMAC and the in and out suffixes the respective inputs and outputs from this supervisory control unit. Zm is the allowable minimum elevation for a particular rotor.

Ideally, these entities need to execute concurrently, which is a natural call for representing each of these entities as a separate thread. However, when looking at the actual implementation of this supervisory control unit in Figure 5.8(b), we see a sequentialization of the concurrency; in particular, the `Supervisor` is executed before the `Monitor`. Simulink toolbox Stateflow was used to implement the design requirement. The parallel *AND states* in Stateflow need a specific ordering to generate C code. This choice of sequentialization may have further ramifications on the implementation in our opinion is not faithful to the specification. On the flip side, modeling these two entities as threads means that the designer must explicitly handle the critical section for $Zout$. Note in Figure 5.8(a) that there is a shared box pointed to with the arrows labeled `update` that eventually sends the $Zout$. Writing the embedded program for this supervisory control unit mandates care in order to get the possible interleaving correct.

5.4.2 A Micro-threading Approach to Concurrent Software Generation

As discussed already, a new version of Polychrony has just been released with multi-threading in place [61]. The approach to multi-threading in the tool is based on signal flow. The number of threads created would be dependent on the number of input and output signals and the dependencies between them. Individual threads are created to read an input, to do the tasks involving that input signal and finally to write the output signal. Every action related to a combination of several inputs would require additional threads. Now these threads are ordered using semaphores which indicate a condition being satisfied for the next thread to start.

A scaled down version of the STARMAC requirement concerning only the elevation monitoring function was modeled in SIGNAL to highlight the concurrency issues. The `Supervisor` and `Monitor` entities are modeled as processes called from a `Control` main process. Both `Supervisor` and `Monitor` processes return their output values back to the `Control` process. There is a deterministic ordering in place to decide who should be updating the output value of $Zout$. Even though the two processes are having different rates, SIGNAL would not allow us to have a random order for update of the output. This would take us back to the original problem in implementation shown in Fig 5.8(b). So a complete parallel implementation in C is not possible with the current multi-threaded model. The SIGNAL code used to model the STARMAC design requirement is as follows:

```
process control = ( ? integer Zin, Zref;
    event tick1, tick2;
 !  integer Zout;  )
```

```
      ( | Zoutsupervisor ^= tick1
        | Zoutmonitor ^= tick2
        | Zoutsupervisor := supervisor (Zin,Zref)
        |  Zoutmonitor:= monitor {1} (Zin)
        | Zout := Zoutsupervisor default Zoutmonitor
        |  )
        where
        integer  Zoutsupervisor init 0, Zoutmonitor init 0;

        process supervisor =
        ( ? integer Zins, Zrefs;
        ! integer Zouts;
         )
            (| Zouts := Zins when (Zins < Zrefs) default Zrefs
             | );

        process monitor =
        { integer Zm}
        ( ? integer Zinm;
        ! integer Zoutm;
         )
            ( | Zoutm := Zinm when (Zinm > Zm)
              |);
end;
```

A thread model based on the threads generated from SIGNAL tool is shown in the Figure 5.9. Here the 3 process example is converted into 7 threads ordered using semaphores. Concurrent functions like read operation for Zin, $Zref$ and $tick1$ are executed in a parallel fashion and the operations on them are performed using multiple threads. Finally the $Zout$ signal is updated using an additional thread. This example clearly illustrates the micro-threading model in Polychrony tool.

In the current strategy, the programmer is kept away from decisions regarding the number of threads and how they should be grouped. In order to utilize multi-core architectures, it is important for the programmer to be aware of the implementation level details, so that he can tailor it according to the hardware resources available to him. Therefore, we suggest an alternative multi-threading solution that may be inferred from the SIGNAL specification, which uses only two threads for the STARMAC example.

5.4.3 Process based Multi-threading Model for SIGNAL Programs

Processes in SIGNAL language due to their modularity are independent entities which communicate under the conditions required for isochrony. Within an endochronous process, SIGNAL

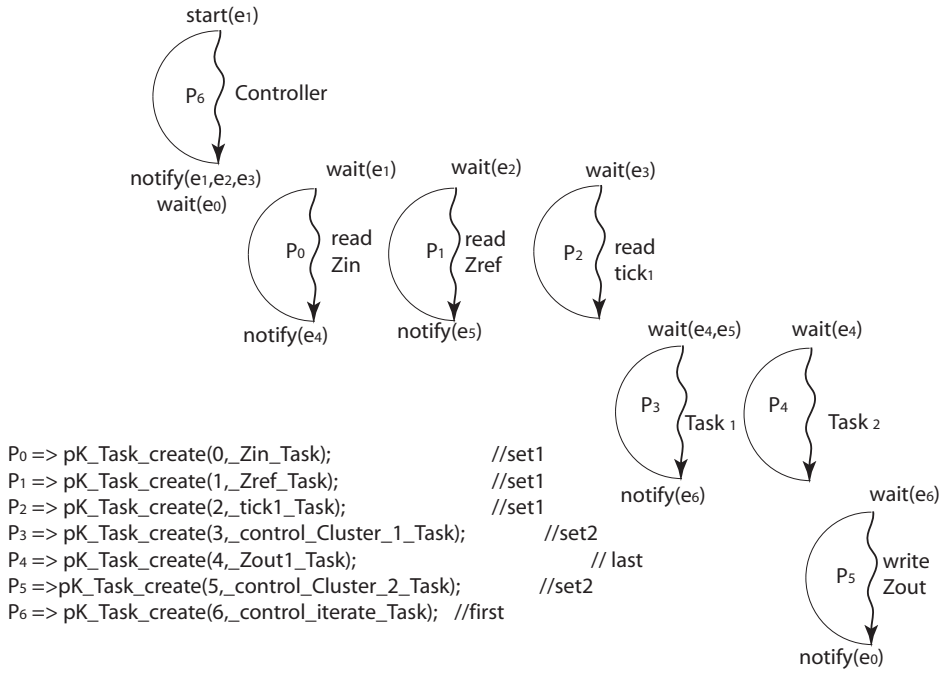


Figure 5.9: Micro-threading Structure for STARMAC

statements are executed in the unique order determined by the flow of data within the process. Taking advantage of the sequential code generation option in the Polychrony tool for modeling the two processes individually and by operate on them with a minimal glue logic, we realize the specification. In order to generate a noninvasive threaded model using SIGNAL, we implemented concurrent processes as individual threads, with a main controller code. Figure 5.10 shows the proposed thread based model, generated as processes using Polychrony tool and joined manually in C. The main controller process has to be executed sequentially and two threads are generated with the two different functions (Supervisor and Monitor) inside. Now these threads can work in parallel, switching between themselves.

During the process of implementing this new model, we first implemented the two functions Supervisor and Monitor as independent SIGNAL programs and observed their generated C code without using the threading option. With the sequential C code for each of the programs as reference, they were converted into a combined code with a controller main calling each as threaded functions. We can observe from the design requirement that the $Zout$ value will be updated by both processes and it needs to be defined in the controller part. The shared variables were all defined in the controller, with the variable needing an update ($Zout$) implemented with Mutex lock. Several inputs well below the minimum value were given as input to verify the functionality of the code. Later the Supervisor process was modified to add more delay into the process. This would mean Monitor process will be updating the $Zout$ variable to ensure correct output. The manual threading inserted into the Polychrony model worked correctly for this particular example. In order to

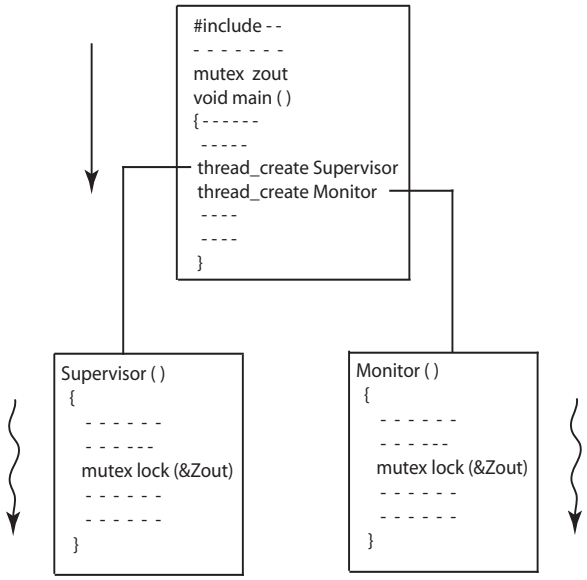


Figure 5.10: Process based Multi-threaded design for STARMAC unit

generate threads directly from Polychrony tool, we need to identify those functions which can be executed from separate threads. Any shared signal would require Mutex lock system associated with it. The amount of rewrite required to generate multi-threaded C code from SIGNAL compiler is minimal. Most of it would be transferring pieces of code from the different files generated from the compiler into the main C file. Now they are addressed as functions and *create-join* constructs are written to access each of them. Mutex lock arrangement is implemented on shared variables for consistency and for avoiding deadlocks.

A producer consumer model with buffer which discussed in Section 5.2.1 was also implemented in SIGNAL. The model contained three processes, namely `Producer`, `Consumer` and `pcmain`. The first two processes communicate with each other using a buffer mechanism designed in the third process. The Polychrony tool generated multi-threaded code containing 17 threads and 17 semaphores for this SIGNAL program. The computation performed by each thread is minuscule and the communication overhead is a greater problem than the distribution of code. It is clear that this multi-threading scheme is an infeasible strategy for bigger models with real-time operation. For scalable solution to this problem, process based multi-threading needs to be applied. The producer consumer example was implemented using three threads, one for each process using our strategy. This model promotes faster sequential execution within each individual process and minimize the synchronization to areas where it is absolutely necessary. The code generated from the Polychrony compiler and the alternate coarse grain code for the two examples discussed here are given in INDEX-B.

The amount of parallelism in the generated code was determined by the compiler for existing multi-threading structures. In the process based technique the control has been given back to designer

to determine the number of processes and consequently the number of distributed nodes/threads in a system. Without any modification to the Polychrony compiler and with minimal glue logic to connect the processes, we have obtained an alternative coarse grain multi-threaded code generator for SIGNAL.

Chapter 6

MRICDF: An alternative polychronous formalism and synthesis methodology

Safety critical applications require embedded software that can guarantee deterministic output and on time results during execution. Synchronous programming languages are by nature focused on providing the programmer with deterministic output at fixed points in time [73]. Synchronous programming languages such as Esterel [43], LUSTRE [74], SIGNAL [71] etc., have successfully generated sequential software for embedded applications targeting avionics, rail transportation, etc. The underlying formalism of these languages is robust and the associated software tools provide correctness preserving refinement from the specification to implementation. But even within the class of synchronous programming languages, they have very distinctive Models of Computation (MoC) [40]. Esterel and LUSTRE use an external global tick (or global clock) as reference for events that occur in the system. SIGNAL has an entirely different Model of Computation, where no assumptions are made on the global tick while designing the system. Each event has its own tick associated with it and the analysis of the specification would have to yield a totally ordered tick for synthesis. This totally ordered tick could be a variable (also known as signal) which has a synchronous event occurring with all events in the system. If there is no such signal in a network the absence of an event, a signal is constructed which has an event happening in synchrony with every event occurring in the system. This signal can be viewed as a root clock of the system with a hierarchical tree structure relating it with every other signal. Since different signal events occur at different rates, there could be signals without an event for a particular time instant. Due to this multi-rate behavior on signals, this MoC is also known as **polychronous** or multi-rate.

The need for polychronous formalism and its advantages over single rate based formalism have been explained in Chapter 1. SIGNAL [66] and its toolset Polychrony [61] provide a framework for multi-rate data flow specification and is the inspiration for our Multi-Rate Instantaneous Channel Connected Data Flow (MRICDF) formalism. However, MRICDF envisions a visual formalism to represent the data flow, with control specification only through *epoch relations* (defined in later sections). The execution semantics and implementability analysis also differ from Polychrony.

Polychrony analyzes the system by solving a set of clock equations [32]. They use ROBDDs for creating canonical form for all clock variables, and then construct clock hierarchy in the form of tree-like data structures. This makes the analysis complex, unnecessarily. MRICDF synthesis methodology seeks to remain in the Boolean theory domain for analysis, but use various techniques such as prime implicate generation for obtaining a total order on events occurring in the network.

6.1 Synchronous structure preliminaries

The semantics of MRICDF is explained in terms of synchronous structures. The preliminaries of synchronous structures are given in [113]. Here we review some of the definitions that are required in explaining the semantics of MRICDF.

Definition 6.1.1 (Events, Signals)**.** *An event is an occurrence of a value. A signal is a totally ordered set of events.*

For a signal x, an event on x is denoted by e_x and the set of all events on x is denoted by $E(x)$. The set of all events in a specification is denoted by Ξ.

Definition 6.1.2 (Synchronous structure)**.** *The pair (Ξ,$\ll$) is a synchronous structure if and only if Ξ is a non-empty set of events and $\ll$ is a preorder on Ξ such that $\forall x \in \Xi \cdot \{y \in \Xi \mid y \leq x\}$ is finite, where the following relations are defined:*

- $(equivalence) \quad x \sim y \Leftrightarrow_{def} x \ll y \wedge y \ll x$
- $(precedence) \quad x < y \Leftrightarrow_{def} x \ll y \wedge x \nsim y$
- $(partial\ order) \quad x \leq y \Leftrightarrow_{def} x < y \vee x \sim y$

With these operators on events in place, we can see that for two events $e_a, e_b \in \Xi$, $e_a \prec e_b$ means that $e_a \preceq e_b$ and $e_a \nsim e_b$. From a code synthesis perspective, a data dependence is a binary relation on events of precedence type. We denote `data dependence` by the notation '$\rightharpoonup$'. So for any two events $e_a, e_b \in \Xi$, $e_a \rightharpoonup e_b$ implies $e_a \preceq e_b$. In the absence of global time reference, an instant or a point in time does not make sense. That is why an instant is defined as the maximal set of events, such that all events in that set are related by '$\sim$' relation with each other. In other words, all events that are synchronized with each other define an instant.

Definition 6.1.3 (Instant)**.** *If the set of all events Ξ is partitioned using the equivalence relation $\sim$, equivalence classes containing events that are synchronous with each other are formed. Each of these equivalence classes is called an instant, i.e., The set of instants $\Upsilon = \Xi / \sim$.*

Any event e_x on a signal x belongs to an instant of a synchronous structure, say $S \in \Upsilon$. We denote it as $e_x \rhd S$. Each set $S \in \Upsilon$ will contain events which have the property $\forall\ e_a, e_b \rhd S$, $a \sim b$. Since instants are equivalence classes, we can define a precedence order between them. For two sets $S, T \in \Upsilon$, $S \prec T$ if and only if, $\forall\ (s,t) \rhd (S,T)$, there exists the relation $s \prec t$.

Definition 6.1.4 (Epoch)**.** *The epoch of a signal is a possibly infinite set of instants where that signal has events.*

For a signal x, $I(x)$ or $\widehat{x}$ are used to denote its epoch. An epoch is also known as *clock* of a signal in SIGNAL terminology. The synchronous relations defined on events in signals are extended as relations between epochs of signals.

Definition 6.1.5 (Synchronous signals)**.** *If two signals have the same epoch , they are said to be synchronous. For two synchronous signals x and y, $I(x) = I(y)$. Both signals have events belonging to the same instant S, where $S \in \Upsilon$. $\forall S \in I(x) \Leftrightarrow S \in I(y)$.*

Definition 6.1.6 (Downsampling)**.** *A downsampling operation on a signal x with epoch $I(x)$ removes events on x, thereby removing instants from its instant set. This creates a new signal y having a lower epoch $I(y)$, i.e. lesser events.*

The epoch of y is said to be a downsampled form of $I(x)$ or $I(y)$ is said to be of smaller epoch, i.e. $I(y) \subset I(x) \subseteq \Upsilon$. $\forall S \in I(y), \exists e_x \in E(x)$ such that $e_x \rhd S$. Conversely, $\exists e_x \rhd T \in \Upsilon$, such that $\forall e_y \in E(y), e_y \not\rhd T$.

Definition 6.1.7 (Upsampling)**.** *An upsampling operation on a signal adds new events to create a new signal with a larger instant set or in other words a higher epoch.*

For a signal x and its upsampled output y, $I(x) \subseteq I(y)$. $\forall S \in I(x), \exists e_y \in E(y)$ such that $e_y \Vdash S$. Also, $\exists e_y \rhd T \in \Upsilon$, such that $\forall e_x \in E(x), e_x \not\rhd T$.

6.2 MRICDF actor network model

Multi-Rate Instantaneous Channel-connected Data Flow (MRICDF) is a data flow network consisting of several actors in a network communicating with each other via instantaneous channels [94]. Each actor in the network will contain input and output ports and a specified computation that will be performed on the input signals. Events in different signals may occur at different instants and there is no external global time reference. As a result, signals may or may not be synchronized with each other. Synchronization information between events may either be implicit in the model or explicitly added to the model, also known as *exogenous constraints*.

Actors could be of two types: **Primitive actors** and hierarchically composed **composite actors**. All actors compute their reaction to a trigger condition in an instant. The communication between actors using an instantaneous-channel is assumed to be completed within an instant. Figure. 6.1 is a simple MRICDF network consisting of 4 actors which communicate through their input and output signals. The MRICDF network here has signals x, y as input and w as output. The intermediate signals are u, v and z. This entire network can be again used as an actor in another MRICDF network. Thus hierarchical specification is enabled. What triggers an actor to react (or

compute) is not a priori fixed, and is computed using epoch analysis. These trigger conditions are derived from the model along with some epoch constraints imposed by the modeler. If they can be derived for all actors in the network, then software synthesis is possible. If not, the user has to add extra constraints to help determine the triggering conditions. A triggering event at any actor could spark a series of computations with event generation at intermediate signals and output interfaces throughout the network.

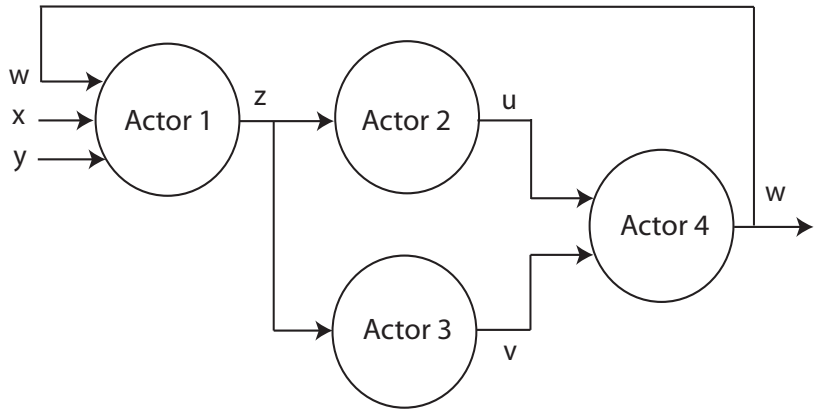


Figure 6.1: An MRICDF Network Model

6.2.1 Primitive and composite actors for MRICDF

MRICDF model is specified in terms of the basic building blocks known as the `primitive` actors. The four primitive actor types are shown in Figure 6.2. Along with each actor a table showing a few cases is shown. $\perp$ is used to denote absence of an event.

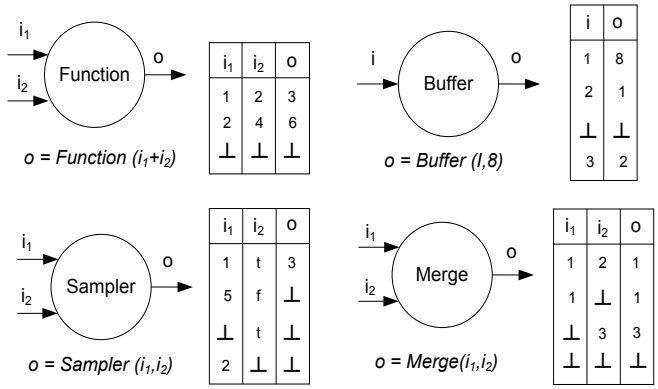


Figure 6.2: MRICDF primitive actors

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. Using a `Function` actor to connect signals requires that these constraints are always maintained. So the input and output signals will have events always for the same instant or in other words, they have the same epoch. The data type of `Function` actor can be integer, Boolean, float etc. For a `Function` actor $F(n, m)$ with input ports i_j where $j = 1..n$, and output ports o_k where $k = 1..m$, $I(i_1) = I(i_2) = .. = I(i_n) = I(o_1) = I(o_2) = .. = I(o_m)$. In Figure 6.2, 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. So at the start of computation an initial value has to be provided for the `Buffer` actor. This is a single-input single-output actor, but it can be used to create a buffer of larger size. The data type at the input and output will be the same and it can be of any type. The input and output events occur within the same instant and hence the epoch constraint associated with `Buffer` actor is $I(i) = I(o)$. In Figure 6.2, 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, $I(o) = I(i_1) \cap I([i_2])$.

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, since the output port contains events from any of the input ports. Hence the epoch equation is $I(o) = I(i_1) \cup I(i_2)$.

A `Composite` actor is an instance of an MRICDF network that was previously designed and reused for the new design. The `Composite` actor will appear as any other actor with its number of input and output ports decided by the number of ports in the reused MRICDF network. Hierarchical specifications can be designed by this approach, since within each `Composite` actor remains another network of primitive or composite actors.

6.2.2 Hierarchical specifications using MRICDF actors

The Producer-Consumer problem or the bounded-buffer problem is a computer science example to demonstrate synchronization problem between multiple processes. We select this example due to its inherent multi-rate nature and for demonstrating hierarchical specification feature of MRICDF. Producer and consumer have separate rates of operation and they must be synchronized for obtaining a read/write from a single-stage buffer. A pseudo code representing the model is shown in Listing 6.1. The SIGNAL representation of this model was given in [97].

Listing 6.1: Pseudo code for Producer-Consumer Model

```
1  Actor Producer (input boolean ptick;
2        output integer dvalue) =
3  (| counter := sampler((prevcount +1) mod 7,ptick)
4   | prevcount := Buffer(counter,0) |)
5
6  Actor Consumer (input boolean ctick,u;
7                          integer bvalue;
8        output integer dread; boolean v) =
9  (| v:= PriorityMerge(Sample(true,u),false)
10  | dread := PriorityMerge(Sample(bvalue,v),-1) |)
11
12 Actor PCmain (input boolean p,c;
13       output integer pcdata, boolean pcvalid) =
14 (| d := Producer(p)
15  | b := PriorityMerge(c, Buffer(d,0))
16  | pc := PriorityMerge(Buffer(p,false),false)
17  | (pcdata, pcvalid) = Consumer(c,pc,b) |)
```

The `Producer` and `Consumer` models are represented as individual composite actors connected to another actor `PCmain` which synchronizes them and also takes care of the storage of data being sent. In the MRICDF representation of the example shown in Figure 6.3, the `PCmain` is separated into multiple composite actors, `Cell` and `Activate`. Figure 6.4 shows the MRICDF composite actor representing the `Producer`. `Producer` actor performs a counter function and its value is being sent to the output on every input trigger represented by *ptick*. So for every event in the signal *ptick*, the `Producer` emits a value at the port *dvalue*.

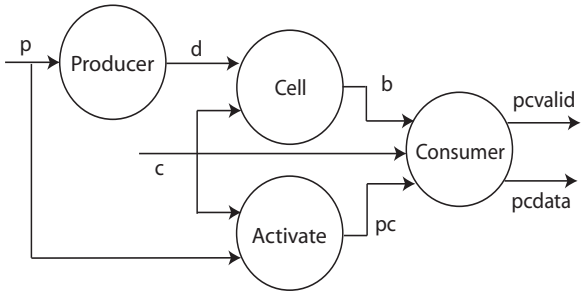


Figure 6.3: Producer Consumer example

The `PCmain` actor in the pseudocode combines both synchronization tasks along with the storage of the generated value. These tasks are separated and shown in Figure 6.5 and Figure 6.6. The

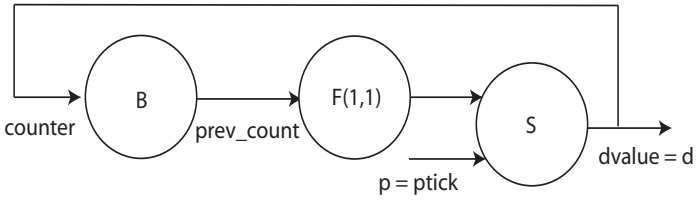


Figure 6.4: Producer actor model

`Consumer` actor is triggered by events arriving at the signal *ctick*. The alias for trigger signals of `Producer` and `Consumer` actors are p and c respectively. The `Activate` MRICDF actor shown in Figure 6.6 accepts these signals and produces a combined trigger for synchronizing all actors. This trigger signal pc denotes instants where there is either a production or consumption of data from the buffer. In the `Consumer` actor shown in Figure 6.7, pc is used to indicate a valid data is present/absent in the buffer. The consumed data is obtained through the port b connected to the storage actor `Cell`. Figure 6.5 shows the structure of this single data storage block. For every `Consumer` tick c, the newly generated value from the consumer d or the value stored in the Buffer is passed onto the output port b. There is a loop with a `Buffer` primitive actor which refreshes the previously stored value. More information about producer-consumer model and other examples implemented using MRICDF are available in [99].

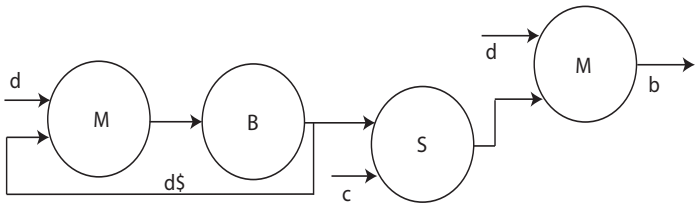


Figure 6.5: Cell actor model

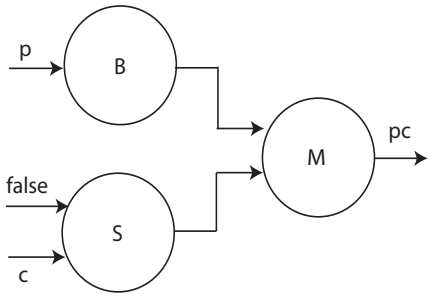


Figure 6.6: Activate actor model

For code synthesis using the clock calculus approach a root clock signal has to be identified. Later a clock tree will be built which has to be hierarchic. Once a unique sequential schedule is decided, code generation can be performed. We now propose an alternative to the clock calculus approach where synthesis can be performed at the Boolean level.

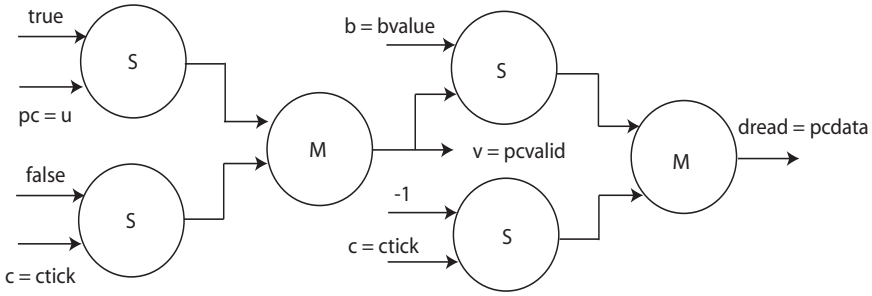


Figure 6.7: Consumer actor model

6.3 Boolean theory approach to embedded software synthesis

The implementation of MRICDF models as embedded softwares involves a detailed analysis of the design and several transformation steps to reach the code generation phase. A correct-by-construction software synthesis methodology has to be put in place to form a total order on all the events occurring in an MRICDF model. Obtaining a unique order of execution is important for sequential software synthesis. Apart from these transformation steps, issues such as causal loop detection have to be handled as well. We propose a Boolean theory approach to software synthesis called Epoch Analysis and propose a prime implicate based technique for implementing it. Finally we will summarize the sequential synthesis conditions for an MRICDF network.

6.3.1 Boolean theory preliminaries

Our approach into 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 or epoch of a signal x as a Boolean variable b_x. Here the Boolean variable contains $true$ 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 $[b_x] \wedge [-b_x] = false$ and also $b_x = [b_x] \vee [-b_x]$.

The epoch relations and Boolean equations representing each MRICDF primitive actor is summarized in Table 6.1. The epoch relations are translated as Boolean equations for `Function`, `Buffer` and `Merge` actors. `Sampler` actor Boolean equations contain additional statements which define $true$ and $false$ valued parts of its second input signal. For implementation purposes, any MRICDF Boolean representation will contain an additional Boolean equation which asserts at least one of the Boolean literals is $true$ for every instant. This is because, the instant where no signal has an event is not an instant to be considered. In the case of these primitive actors, a Boolean equation $b_a + b_b + b_c = true$ (in case of `Buffer`: $b_a + b_b = true$) is included to force at least one of the signals to have a $true$ value. This avoids the trivial case where all signals are $false$.

MRICDF Actor	**Epoch relation**	**Boolean Equations**
Function $F(2,1)$	$\widehat{a} = \widehat{b} = \widehat{c}$	$b_a = b_c, b_b = b_a$
Buffer B	$\widehat{a} = \widehat{b}$	$b_a = b_b$
Sampler S	$\widehat{a} = \widehat{b} \cap [\widehat{c}]$	$b_a = b_b \wedge [b_c], b_c = [b_c] \vee [\neg b_c], [b_c] \wedge [\neg b_c] = false$
Merge M	$\widehat{a} = \widehat{b} \cup \widehat{c}$	$b_a = b_b \vee b_c$

Table 6.1: MRICDF actor primitives and their epoch constraints

A Boolean theory is formed by a set of boolean equations that represent the relationships between signals. Our synthesis technique requires computation of prime implicates of this theory. Hence we provide their definitions as well. Max terms 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 6.3.1 (Implicate)**.** *An implicate is defined as a sum term that can cover max terms of a function.*

Definition 6.3.2 (Prime Implicate)**.** *A prime implicate is defined as a sum term that is not covered by another implicate of the function.*

A survey of prime implicate finding algorithms can be found in [81]. [56] used a data structure known as *tries* to improve the performance of prime implicate generators. We use a prime implicate generator developed by University at Albany based on their custom *prime implicate tries* [109].

6.3.2 Epoch Analysis: Boolean theory approach to sequential implementability

The issues in transformation of an MRICDF network involve causal loop detection and the identification of a deterministic order of events in an MRICDF network. In Polychrony, synthesis condition (Defn. 6.3.3) is to identify an acyclic, hierarchic clock tree where the root node represents the signal with highest clock or epoch [132]. In polychronous formalism, acyclic specification is one which is devoid of causal loops.

Definition 6.3.3 (Polychrony-Sequential synthesis)**.** *A SIGNAL program with a process P is compilable if it is well-clocked and acyclic.*

Akin to hierarchic 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 (or root clock) and a follower set (a unique order of execution). Epoch analysis provides a Boolean theory approach to check for

sequential implementability of an MRICDF network, i.e. identify the signal with highest epoch and check for the existence of a unique order of computation for the MRICDF network. Together, causality analysis and sequential implementability will be the MRICDF sequential synthesis condition.

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 6.3.4 (Topological Loop)**.** *A polychronous specification is said to contain a topological loop provided the specification consists of a set of actors (i) $a_1, a_2, .., a_n$, such that an output port of a_n is connected to an input port of a_1; and (ii) for all actors $a_1, a_2, .., a_{n-1}$, $\forall i \in \{1..(n-1)\}$, an output port of the a_i^{th} actor is connected to an input port of the a_{i+1}^{th} actor.*

Definition 6.3.5 (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 6.3.6 (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 an instant where all signals in the topological loop have an event. For the instant $S \in \Upsilon$ where apparent causal loop exists: $\forall_{i=1}^{n}, \exists e \in E(x_i) \wedge e \rhd S$.*

An algorithm for causality would look for buffer-free loops initially and then evaluate Boolean equations to check if a causal loop can be constructed. Accordingly, an MRICDF network is rejected or forwarded for sequential implementability checks. The existing causality analysis techniques have been covered in Chapter 3. We will explain more about our causality analysis contributions in Chapter 9.

Existence of Master Trigger signal

A sequentially implementable polychronous model is *endochronous* in nature. Endochronous specifications have a unique time line with a totally ordered set of instants. As explained in Section 3.3.1, we need to identify a signal with the same instant set to act as a reference signal known as the Master trigger signal.

Definition 6.3.7 (Master Trigger [93])**.** *Let M be an MRICDF model. Let 't' be a signal with $E(t)$ as its set of events and Υ the set of all instants of the MRICDF network. For each $S \in \Upsilon$, if there exists an event $e_t \Vdash S$, then '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 6.1. The system of Boolean Equations F defines a theory Σ which is the set of all satisfying assignments for the system. A disjunctive clause C belonging to the theory Σ 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 implicates 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 6.3.1 (Test for Master Trigger signal [93])**.** *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.

Proof sketch: Recall that a master trigger is one which has one event in every instant of the set of abstract instants Υ. The instants are constructed by partitioning the union of events from all signals. Thus if x has to have an event in every instant, its instant set must be a superset of all instants in the model. In other words, every signal's instant set must be a subset of its instant set. By definition b_x should then be implied by every variable b_y for all signals y in the system. b_x encodes the presence of an event of x in an arbitrary instant T, and if $b_y \rightarrow b_x$, then if y has an event in T, so does b_x. So if the solution set of this equation system implies that for any signal y in M, $b_y \rightarrow b_x$, then when we set b_x to false, all left hand sides of those implications have to be set to false. ■

Corollary 6.3.2 (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 6.1 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. 6.3.2 is an extension of **Thm. 6.3.1** 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. 6.3.2** is used.

The conversion of MRICDF actor to its Boolean form for master trigger identification test is shown in Fig. 6.8. Each primitive actor is first expressed in terms of its Boolean equations from Table 6.1 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. 6.8, 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. 3.5 and `Merge` actor. The endochronization of `Merge` actor is discussed along with the follower set generation process in the following section.

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. 6.3.1**. 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 with no unique order of execution or in SIGNAL terminology those with unconnected clock trees.

Definition 6.3.8 (Follower set)**.** *Given an MRICDF model M and its Boolean system of equations B_M, let the follower set of M be FS_M. The first element of FS_M (denoted as FS_M^{e0}) contains signals satisfying **Thm. 6.3.1** for B_M and are data dependent only on signals that are part of FS_M^{e0}. The Boolean system B_{Mi} to find each follower set element FS_M^{ei} is generated by setting Boolean signals in $FS_M^{e(i-1)}$ to $true$ in the system $B_{M(i-1)}$. The successive elements of follower set FS_M^{ei}, contain signals satisfying **Cor. 6.3.2** and have an explicit Boolean equation in B_{Mi} containing only*

BUFFER	FUNCTION	SAMPLER
a = Buffer b Boolean equations 1) $b_a = b_b$, 2) b_a+b_b = true Boolean Function F_B = ((b_a*b_b) +($b_{a'}$ * $b_{b'}$)) * (b_a+b_b) CNF 1:b_a, 2:b_b p sat 2 (*(+(*(1 2) *(-1 -2)) +(1 2))) Prime Implicates:{b_a},{b_b} Master Trigger: {b}	a = Function b c Boolean equations 1) $b_a = b_b$, 2) $b_a = b_c$ 3) $b_a+b_b+b_c$ = true Boolean Function F_F = ((b_a * b_b) + ($b_{a'}$ * $b_{b'}$)) * ((b_a * b_c) + ($b_{a'}$ * $b_{c'}$)) * ($b_a+b_b+b_c$) CNF 1:b_a, 2:b_b, 3:b_c p sat 3 (*(+(*(1 2) *(-1 -2)) +(*(1 3) *(-1 -3)) +(1 2 3))) PI: {b_a},{b_b},{b_c} Master Trigger: {b},{c}	a = b Sampler c Boolean equations 1) $b_a = b_b \cap b_{[c]}$, 2) $b_a+b_b+b_c$ = true, 3) $b_c = b_{[c]} \cup b_{[-c]}$, 4) $b_{[c]} \cap b_{[-c]}$ = false Boolean Function F_S = ((b_a * (b_b*$b_{[c]}$)) + ($b_{a'}$ * ($b_{b'}$+$b_{[c]'}$))) *((b_c*($b_{[c]}$*$b_{[-c]'}$)) + ($b_{c'}$*($b_{[c]'}$+$b_{[-c]'}$))) *($b_{[c]'}$ + $b_{[-c]'}$) *($b_a+b_b+b_c$) CNF 1:b_a, 2:b_b, 3:b_c, 4:$b_{[c]}$,.. p sat 5 (*(… …)) PI: {(b_b+b_c)},{b_b,$b_{[c]}$,$b_{[-c]}$} Master Trigger: {}

Figure 6.8: Master trigger identification for MRICDF primitive actors

signals that are part of $FS_M^{e(i-1)}$.

Endochronous actors `Buffer` and `Function` have signals of equal epoch and hence their follower sets have only a single element: $\{\{a, b\}\}$ and $\{\{a, b, c\}\}$ respectively. The non-endochronous `Merge` when put to the master trigger identification test returns only one PI candidate which is not an output signal ($(b_b + b_c)$, see Fig. 6.9). 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. 6.9. 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. 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. 6.9. 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 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. 6.3.2**) and hence a PI result $(b_b + b_c)$ is valid with $(b \vee c)$ 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 $F_M = \{\{x, y\}, \{(b \vee c), ([x] \vee [y]), a\}\}$. This is equivalent to the clock tree shown in Fig. 6.9 at the bottom with x, y as root clock and b, c as child nodes.

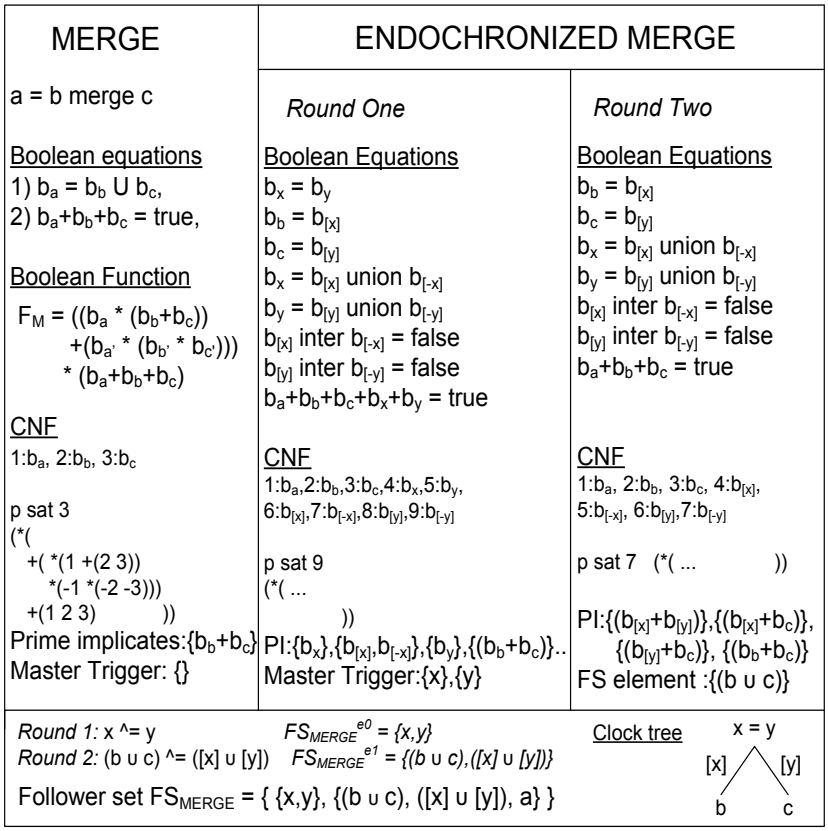


Figure 6.9: Endochronized Merge actor with its follower set

Tests for sequential implementability check of an MRICDF network has been discussed so far. Causality analysis conditions have been explained, but the tests for causality analysis will be explained in Chapter 9. Now we summarize the sequential synthesis conditions for MRICDF networks.

Definition 6.3.9 (MRICDF - sequential synthesis)**.** *Sequential synthesis of MRICDF specification requires that, i) there exists no apparent causal loop in the MRICDF network (**Defn. 6.3.6**); ii) a master trigger signal can be found by applying the master trigger identification procedure*

*(**Thm. 6.3.7**) on signals in the network; and iii) a follower set can be generated according to **Defn. 6.3.8** which contains all signals in the network.*

We have now discussed the Boolean approach to synthesis as an alternative to the clock calculus approach of Polychrony. Epoch Analysis, the Boolean approach to sequential software synthesis was implemented in EmCodeSyn synthesis tool. EmCodeSyn will implement using prime implicate based synthesis technique, the specifications that satisfy the sequential synthesis conditions put forth by **Defn. 6.3.9**.

Chapter 7

EmCodeSyn: Visual framework for software synthesis from MRICDF specification

Programming using single-clocked languages, having an external reference of time, is more natural for users and it follows conventional software synthesis methodology. Hence synchronous languages such as Esterel and LUSTRE are relatively more popular than polychronous formalism based language SIGNAL. Consequently, code synthesis tools like Esterel Studio [131] and SCADE [62] are being used in the industry more often than Polychrony [61]. Multi-Rate Instantaneous Channel Connected Data Flow (**MRICDF**) is a graphical data flow formalism which intends to give a visual perspective to synchronous languages [94]. MRICDF aims to replace the clock equations with *epoch equations* implied by data flow, which are more familiar to engineers. They also inherit the properties of data flow concept which makes MRICDF more modular and suited for composition. We aim to implement Boolean theory approach to software synthesis from MRICDF specifications using a prime implicate based methodology and provide a GUI with easy design and debug facilities for synthesizing MRICDF models. Chapter 6 described the transformation of MRICDF models to Boolean equations and the prime implicate technique for finding master trigger and follower set. In this chapter, we explain the EmCodeSyn framework and the transformation from an MRICDF network to C code using a case study.

7.1 Design Methodology of EmCodeSyn

One of the goals of MRICDF is to popularize the use of polychronous formalism among engineers in the embedded software field. Embedded Code Synthesis or **EmCodeSyn** is a software synthesis tool which provides an environment to specify MRICDF models and utilities to perform the debugging and code generation from specification [88]. The design methodology of EmCodeSyn is

shown in Figure 7.1. The MRICDF model specified using the GUI is stored in a Network Information File (NIF). This will contain information about the types of actors, their inter-connections, computation to be performed, and other GUI information about placement of actors, lines drawn, etc. Now Epoch Analysis is performed on the model which will breakdown the composite actors in terms of primitive actors and translate the model into epoch equations and its corresponding Boolean equations as explained in Section 6.3.1. Now master trigger identification tests based on Theorem **??** is performed to identify a reference trigger for each round of computation. If a master trigger cannot be found within the system, exogenous rate constraints will have to be provided to force a master trigger on the MRICDF model. Later follower set computation is performed which is a sequential schedule for updating values of signals in the MRICDF network. Once a follower set is established for the MRICDF model, code generation can begin. There are three C files generated by EmCodeSyn that can be executed using any generic C compiler. The executed code reads from text files for each input signal and provides a consolidated output file containing all signal outputs with respect to the master trigger events. Thus the behavior of the MRICDF network is simulated using the generated C code.

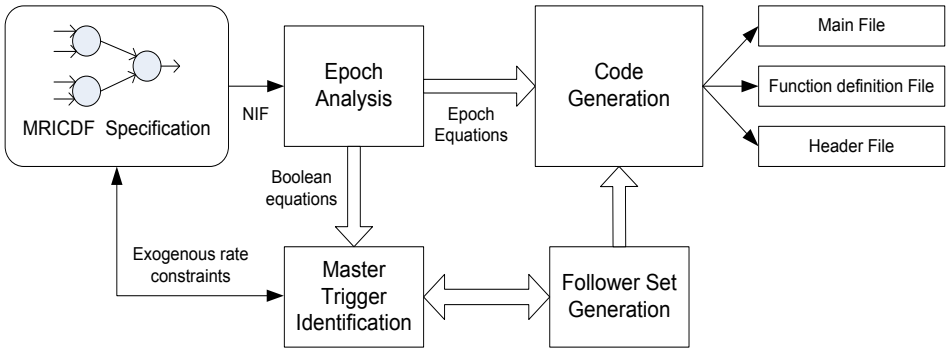


Figure 7.1: EmCodeSyn Design methodology

7.2 Transformation of an MRICDF network into sequentially implementable code

We use a polychronous specification `Absolute` to illustrate the stages in the transformation of an MRICDF network to C code. The intention of the program is to find the absolute value of the given integer input. The polychronous specification $Absolute$ in the programming language SIGNAL is shown in Listing 7.1. 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 $absol$. It performs this operation by checking if the input is less than zero, denoted by the signal neg. On $true$ values at neg, the input value multiplied by -1 ($mult$) is copied to the output signal $absol$. Otherwise, the input value $inval$ is sent to the output signal.

Listing 7.1: Endochronous SIGNAL program `Absolute`

```
1  process absolute =
2     ( ? integer inval;
3       ! integer absol;       )
4     (| neg := true when (inval < 0  )
5                   default false when (inval >= 0)
6      | mult := (inval * -1)
7      | absol:= (mult when neg) default
8                           (inval when (not neg))|)
9      where boolean neg;  integer mult; end;
```

The SIGNAL program is visualized as an MRICDF network with 8 actors in Figure 8.3. The two `Sampler` actors `S1` and `S3` have constant values $true$ and $false$ respectively at their first input port. The `Function-Sampler` actor combination `F1-S1` and `F3-S3` computes its SIGNAL equivalent $when(inval < 0)$ and $when(inval \geq 0)$ respectively. $mult$ is computed by `F2`, while the presence of a negative value is decided by neg, the output port of actor `M1`. The final output $absol$ is the output of the `Merge` actor `M2`.

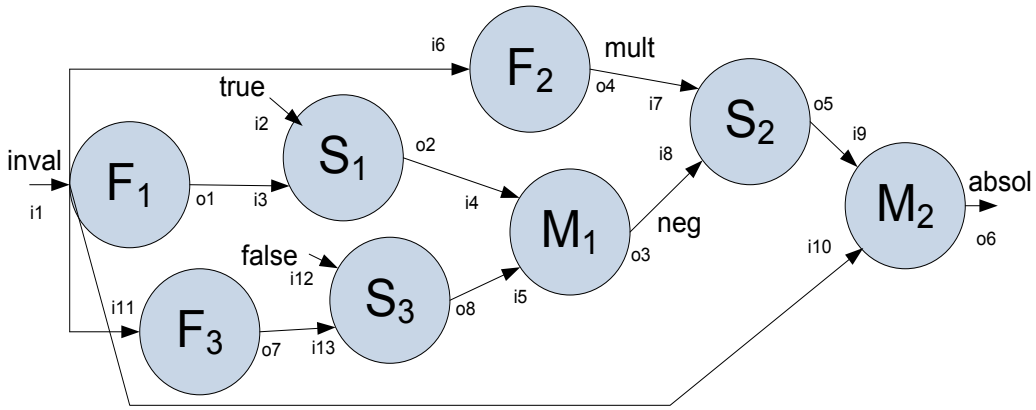


Figure 7.2: The MRICDF network `Absolute`

7.2.1 Environment description of EmCodeSyn

EmCodeSyn is a visual software environment for the design, debug and verification of MRICDF models. The EmCodeSyn environment with the MRICDF example `Absolute` is shown in Fig. 7.3. The Graphical User Interface (GUI) of EmCodeSyn is constructed using Qt. Qt is an application framework with an intuitive API and a rich C++ library for GUI development [13]. In the EmCodeSyn environment, buttons are provided to select actors, port connections and interconnecting lines to form the MRICDF networks. Apart from various primitive actor types, an existing MRICDF network can be saved as a composite actor to be reused in another MRICDF design. Additional model inputs such as epoch constraints can be given using the window the

bottom. Console and error windows are provided to obtain output information. Apart from design utilities, the environment provides facilities to perform code generation. `Follower Set` and `Generate code` are some of the buttons which perform the respective functions for an MRICDF network. There are other facilities included in EmCodeSyn, such as `Causal loop check`, `Actor Elimination`, `MRICDF-SMT` or `MRICDF-Quartz` generation which will be discussed in the following chapters.

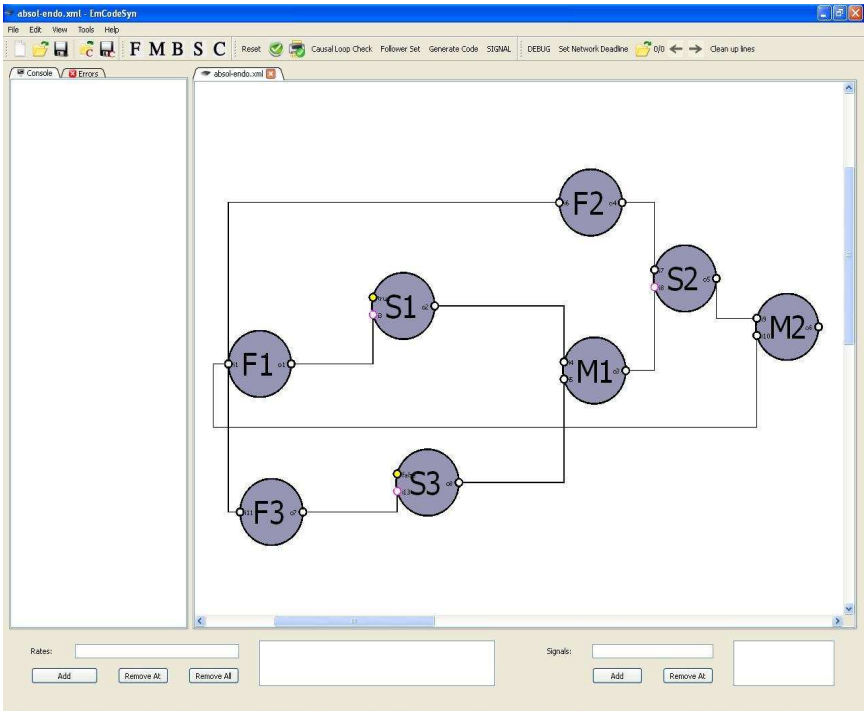


Figure 7.3: EmCodeSyn GUI with an MRICDF network

While specifying MRICDF actors, the data type of its input-output ports need to be mentioned. The data type matching of interconnected actors are checked during the design process to avoid later errors. While instantiating a Function actor, the function it represents has to be specified along with it. This code has to be provided in a dll file for use in the code generation stage. Once the design has been completed or when it is saved, an XML scheme is followed to generate a Network Information File (NIF) to store information. The NIF will contain the information about actor types, the connection details of input and output ports and the type of each port. The contents of the NIF for the network `Absolute` is given in Listing 7.2. Under the root element 'network absoulte', the actors are stored as child elements. In Listing 7.2, only the details relating to actors `F2, M2,` and `S2` are shown due to space constraints. The subchild fields of each actor stores the information about input-output ports and functionality under the fields 'input', 'output' and 'actorDescription'. There is a means to schedule actors according to their execution times and deadlines, but they are not available for polychronous specifications. Hence we do not describe its

details in this chapter. Primitive actors such as `M2` and `S2` have a standard description in the XML file based on their data type. Description of a Function actor contains the associated C code to be plugged in during code generation stage. Other than the actor details, connections between actors, explicit epoch constraints, position information of actors and lines are also part of the NIF of an MRICDF network.

Listing 7.2: Network Information File for MRICDF network `Absolute`

```
<network absolute>
    <actor execution="1" deadline="10" actorID="F2">
        <input inID="6" inType="0" inName="a">
            <equalConnection CON="F1.i.1"/>
        </input>
        <output outID="4" outType="0" outName="mult">
            <outConnection CON="S2.i.7"/>
        </output>
        <actorDescription>
            <Primitive SignalCode="" FuncName="int (int a);
             " WhatKind="Function"
               FuncDef="int Synth::(int a)&#xa;{ return a*-1"/>
        </actorDescription>
    </actor>
    <actor execution="0" deadline="100" actorID="S2">
        <input inID="7" inConnection="F2.o.4" inType="0"/>
        <output outID="5" outType="0">
            <outConnection CON="M2.i.9"/>
        </output>
        <input inID="8" inConnection="M1.o.3" inType="-1"/>
        <actorDescription>
            <Primitive WhatKind="Sampler"/>
        </actorDescription>
    </actor>
    <actor execution="0" deadline="100" actorID="M2">
        <input inID="9" inConnection="S2.o.5" inType="0"/>
        <input inID="10" inType="0">
            <equalConnection CON="F1.i.1"/>
        </input>
        <output outID="6" outType="0" outName="absol"/>
        <actorDescription>
            <Primitive WhatKind="Merge"/>
        </actorDescription>
    </actor>

    ...
    <!--////////// OTHER ACTOR DETAILS ///////-->
    ...

    <!--////////////////////////////////////-->
    <!--///////Item Position Information/////-->
    <!--////////////////////////////////////-->
    <actorItem execution="1" functionDefinition="int Synth::(int a)&#xa;
    { return a*-1" SignalCode="" actorType="0" deadline="10"
                  id="F2" function="int (int a);" yPos="355" xPos="742"/>
    <actorItem execution="0" actorType="3" deadline="100" id="S2" yPos="433" xPos="941"/>
    <actorItem execution="0" actorType="1" deadline="100" id="M2" yPos="486" xPos="1119"/>
    ...
    <!--//////////OTHER ACTORS POSITIONS ///////-->
    ...
    <connectionItem constant="" rotDeg="0" type="0" id="i1" parent="F1"
                name="inval" subConnections="" yPos="-0" xPos="-65"/>
    <connectionItem constant="" rotDeg="0" type="1" id="o1" parent="F1"
                subConnections="" yPos="-0" xPos="65"/>
```

```
55        ...
56        <!--//////////OTHER ACTORS CONNECTIONS //////-->
57        ...
58
59        <connectionItem constant="" rotDeg="0" type="1" id="o8" parent="S3"
60                        subConnections="" yPos="-0" xPos="65"/>
61        <lineItem xStartPos="438" id="c8" xEndPos="470.2148132324219"
62                        yEndPos="480.8232421875" yStartPos="525"/>
63        <lineItem xStartPos="677" id="c11" xEndPos="274"
64                        yEndPos="365" yStartPos="355"/>
65        ...
66        <!--//////////OTHER LINE POSITIONS //////-->
67        ...
68        <lineItem xStartPos="817" id="c5" xEndPos="878"
69                        yEndPos="447" yStartPos="527"/>
70  </network>
```

7.2.2 Epoch Analysis of an MRICDF network

The epoch analysis of `Absolute` consists of checking for the presence of any causal loops and the identification of master trigger signal and follower set. Causal loop analysis is performed by a search algorithm while master trigger and follower set computation is performed using the prime implicate technique. The Epoch equations of MRICDF network `absolute` is shown in Listing 7.3 as per 6.1. Each epoch equation is representative of the actor type and their connections. The `Sampler` actors (say `S1`) are accompanied by two more epoch equations which define the second port which is of Boolean type (here `S1.i.3`). Since $true$ value at `S1.i.3` denoted as `[S1.i.3]` is required for firing `S1`, the Boolean relations connecting `[S1.i.3]`, `[-S1.i.3]` and `S1.i.3` have to be considered. They are given after each `Sampler` epoch equation in Listing 7.3. The reasoning for each Boolean equation has been discussed in Section 6.3.1. The type of each epoch equation is given at the left hand side using a number which is useful for parsing. After all the actor epoch equations, the actor connections are provided which equates their signal epochs.

Listing 7.3: Boolean equations for MRICDF network `Absolute`

```
1   0 F1.o.1 = F1.i.1                    // F1
2   2 S1.o.2 = S1.i.2 & [S1.i.3]         // S1
3   1 S1.i.3 = [S1.i.3] + [-S1.i.3]      // S1.i.3
4   5 -[S1.i.3] + -[-S1.i.3]             // S1.i.3
5   0 F2.o.4 = F2.i.6                    //F2
6   2 S2.o.5 = S2.i.7 & [S2.i.8]         //S2
7   1 S2.i.8 = [S2.i.8] + [-S2.i.8]      //S2.i.8
8   5 -[S2.i.8] + -[-S2.i.8]             //S2.i.8
9   1 M2.o.6 = M2.i.9 + M2.i.10          //M2
10  0 F3.o.7 = F3.i.11                   //F3
11  1 M1.o.3 = M1.i.4 + M1.i.5           //M1
12  2 S3.o.8 = S3.i.12 & [S3.i.13]       //S3
13  1 S3.i.13 = [S3.i.13] + [-S3.i.13]   //S3.i.13
14  5 -[S3.i.13] + -[-S3.i.13]           //S3.1.13
15  0 F1.o.1 = S1.i.3
16  0 S1.o.2 = M1.i.4
17  0 F2.o.4 = S2.i.7
18  0 S2.o.5 = M2.i.9
```

```
19  0 F3.o.7 = S3.i.13
20  0 M1.o.3 = S2.i.8
21  0 S3.o.8 = M1.i.5
22  0 F2.i.6 = F1.i.1
23  0 F2.i.6 = M2.i.10
24  0 F2.i.6 = F3.i.11
```

After epoch equations are generated, the prime implicate generator is used to find the master trigger signal. The MRICDF network is converted from Boolean form to Conjunctive Normal Form (CNF). The CNF array of Boolean variables and the CNF form of the `Absolute` MRICDF network is shown in Listing 7.4. The transformation of Boolean equations into CNF form for each actor type has been discussed in Section 6.3.2. The 27 variable Boolean function representing the MRICDF design is shown in the listing. Multiple single positive literals are reported as prime implicates, as shown at the bottom Follower Set (1) in Listing 7.4. The input signal $inval$ (`F1.i.1`) is one of them and is fit to be a master trigger. It can be computed from input signals or initial values in a `Buffer` actor. Other prime implicates reported are also part of the follower set first element such as `S1.i.3,F2.i.6`, etc. It can be observed that all of these signals are of the same epoch. The second element of the follower set FS_{absol}, will consist of signals of lesser epoch such as $(M1i4 \vee M1i5)$, 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 $FS_{absol} = \{\{inval\}, \{(M1i4 \vee M1i5)\}, \{M2i9\}\}$. The other signals having the same epoch as of the representative element in FS_{absol} is shown in Listing 7.4.

Listing 7.4: CNF equations for MRICDF network `Absolute`

```
1  c SAT File - Equation conversion from ./Generated_Files/absolute-endo/absolute-eq.txt
2  c Variables {n, (n+1), (n+2)} where (n mod 3) == 1 correspond to
3  c {vk,[vk],[~vk]} where k=(n-1)/3
4  p sat 27
5  (*(
6     +( *(2 1) *(-2 -1) )
7     +( *(7 *(3 5)) *(-7 +(-3 -5)) )
8     +( *(4 +(5 6)) *(-4 *(-5 -6)) )
9     +(-5 -6)
10    +( *(9 8) *(-9 -8) )
11    +( *(14 *(10 12)) *(-14 +(-10 -12)) )
12    +( *(11 +(12 13)) *(-11 *(-12 -13)) )
13    +(-12 -13)
14    +( *(17 +(15 16)) *(-17 *(-15 -16)) )
15    +( *(19 18) *(-19 -18) )
16    +( *(22 +(20 21)) *(-22 *(-20 -21)) )
17    +( *(27 *(23 25)) *(-27 +(-23 -25)) )
18    +( *(24 +(25 26)) *(-24 *(-25 -26)) )
19    +(-25 -26)
20    +( *(2 4) *(-2 -4) )
21    +( *(7 20) *(-7 -20) )
22    +( *(9 10) *(-9 -10) )
23    +( *(14 15) *(-14 -15) )
24    +( *(19 24) *(-19 -24) )
25    +( *(22 11) *(-22 -11) )
26    +( *(27 21) *(-27 -21) )
27    +( *(8 1) *(-8 -1) )
28    +( *(8 16) *(-8 -16) )
29    +( *(8 18) *(-8 -18) )
```

```
30     +(1 2 3 4 5 7 8 9 10 11 12 14 15 16 17 18 19 20 21 22 23 24 25 27)
31  ))
32
33  CNF Array:
34  [0] = blank [1] = F1.i.1 [2] = F1.o.1 [3] = S1.i.2 [4] = S1.i.3 [5] = [S1.i.3]
35  [6] = [-S1.i.3] [7] = S1.o.2 [8] = F2.i.6 [9] = F2.o.4 [10] = S2.i.7 [11] = S2.i.8
36  [12] = [S2.i.8] [13] = [-S2.i.8] [14] = S2.o.5 [15] = M2.i.9 [16] = M2.i.10
37  [17] = M2.o.6 [18] = F3.i.11 [19] = F3.o.7 [20] = M1.i.4 [21] = M1.i.5
38  [22] = M1.o.3 [23] = S3.i.12 [24] = S3.i.13 [25] = [S3.i.13] [26] = [-S3.i.13]
39  [27] = S3.o.8
40
41
42  Follower Set:
43  (1) {F1.i.1 } , *{S1.i.3}* ,  *{F2.i.6}* , *{S2.i.7}* ,   *{M2.i.10 }* ,
44      *{F3.i.11 }* , *{S3.i.13}*
45  (2) {[S1.i.3] + [S3.i.13]} , *{[S1.i.3] + M1.i.5}* , *{S2.i.8}* ,
46       *{M1.i.4 + M1.i.5}* , *{M1.i.4 + [S3.i.13]}*
47  (3) {[S2.i.8]} , *{M2.i.9}*
```

7.2.3 Code generation for an MRICDF network

Code generation process starts when the follower set generation is complete. Each iteration in the generated code is triggered by a new event on the master trigger signal and the order of execution is determined by the follower set and the data flow obtained from the Network Information File. The generated code is placed in three C files: Header file, Function Definition file and the Main file. The Header file contains a structure with all the functions and primitives that are to be used. The Function Definition file contains the code for each function declared in the header file, including those for the primitive ones. The Main file is the backbone of the synthesized output, containing the order of execution.

Each of the primitive actor types with varied input-output data types is stored in the Function definition file and Header file. Figure 7.4 shows the plugged-in code for standard MRICDF actors. The `Merge` and `Sampler` code has a standard format, but the `Function` actor is customized according to the user's need. The `Buffer` actor alone is not placed in the Function Definition file. It is present in the Main file, since it acts as a source for input events at times. We can see in Figure 7.4, the `Buffer` code sets the Boolean output trigger variable 'a.B1OS1' to $true$ or $false$ based on input events at the input trigger variable 'a.B1IS1'. Each of these functions are called from the Main file when their firing conditions need to be evaluated due to input events.

The Main file for an MRICDF network consisting of a single `Merge` actor of Boolean type is given in Figure 7.6. First the input files are read and the presence of new events at master trigger signal is checked inside a continuous 'while(true) loop'. In the case of a `Merge` actor, an external reference clock is required. On $true$ events at master trigger, the clock files of both input ports are read. This is indicated by the line 'a.M1i1_clock' and 'a.M1i2_clock'. If there are events at input port, a boolean variable is set to indicate the trigger. Finally the `Merge` actor function is called with the input-output signals and the trigger variables as arguments. This iterative loop is continuously executed until, the master trigger has no more input events.

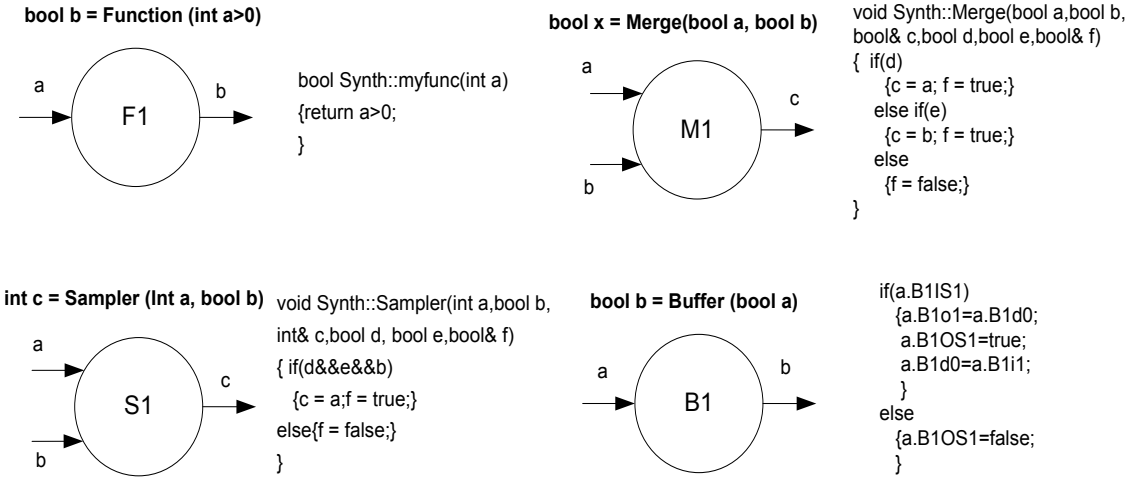


Figure 7.4: Primitive actor code in Function definition file

Once the C files are obtained from EmCodeSyn, any C compiler can be used to execute the code. A sample input trace for the `Merge` MRICDF network is shown in Figure 7.5(a). The clock files for input ports `M1i1` and `M1i2` are read from `c_M1i1.txt` and `c_M1i2.txt` respectively. Whenever there is an input event at a port (denoted by 1 in clock file), the next input value is read from a value file provided for each input (v_M1i1.txt and v_M1i2.txt). Once the code is executed, an output file is produced which is shown in Figure 7.5(b). The presence and absence of events are shown by the input values or $\perp$ symbol respectively. The `Merge` functionality can be observed at the output port column `M1o1`.

c_M1i1	v_M1i1	c_M1i2	v_M1i2
1	t	0	f
0	t	1	f
0	t	0	t
1	f	1	t
1	f	0	f
1	f	1	f
0	t	0	t

(a) Input values for `Merge` actor

tick#	M1i1	M1i2	M1o1
0	t	$\perp$	t
1	$\perp$	f	f
2	$\perp$	$\perp$	$\perp$
3	t	f	t
4	t	$\perp$	t
5	f	t	f
6	$\perp$	$\perp$	$\perp$

(b) Output file for `Merge` actor

Figure 7.5: Output from execution of generated code

Now let us analyze code generation for the MRICDF network `Absolute`. The significant parts of

```
#include "Merge_pib_syn_header.h"
int main()
  {
     Synth a;
     string line;
     if(a.Read_Input_Files()==false){return 1;}
     int iteration(0);
     a.Send_Output_Header();
     while(true)
     {
      a.Update_InOutSent(); //Resets all OutSent variables to false
      if(a.clkfile_M1i1.eof()) // READ M1i1 clock
        {return 0;}
      getline(a.clkfile_M1i1,line);
      a.M1i1_clock = atoi(line.c_str());

      if(a.M1i1_clock)
                  {if(a.vfile_M1i1.eof()) // Reach M1i1 values
                    {return 0;}
                   getline(a.vfile_M1i1,line);
                   if(line=="true")
                     {a.M1i1 = true;}     // M1i1 has value true
                   else if(line=="false")
                     {a.M1i1 = false;}    // M1i1 has value false
                   a.M1IS1 = true;        // New event at M1i1 port
                  }
                  else
                  {
                   a.M1IS1 = false;       // No event at M1i1 port
                  }

      if(a.clkfile_M1i2.eof()) // READ M1i2 clock
        ...
        ...                  // Similar code for M1i2 port

      a.Merge(a.M1i1,a.M1i2,a.M1o1,a.M1IS1,a.M1IS2,a.M1OS1);
      a.Send_Output_To_File(iteration);
      iteration++;
     }
    return 0;
   }
```

Figure 7.6: Main file for `Merge` actor

the the main and function definition files for `Absolute` are shown in Fig. 7.7. The main.cpp contains an 'while(true)' loop limited to the number of input events provided from the environment. Within this loop ($SECTION\ 1$), 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. Each actor input port has a Boolean variable associated with it which denotes the presence or absence of the associated signal event ($F1IS1$ for `F1` input, $F3IS11$ for `F3` input, and so on). $inval$ events alone are read 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 (`Merge` Boolean, `Merge` Integer, `Sampler` Boolean, etc.) is included along with custom `Function` actors such as `fc1less`.

absolute_pib_syn_main.cpp

```
#include "absolute_pib_syn_header.h"
int main()
{   Synth a; string line;
    if(a.Read_Input_Files()==false){return 1;}
    int iteration(0);
    a.Send_Output_Header();
     while(true)
     {
      a.Update_InOutSent(); //Resets all OutSent variables to false
      //******SECTION 1******READ master trigger events & inputs
      if(a.clkfile_F1i1.eof())      {return 0;} // READ M1i1 clock
      getline(a.clkfile_F1i1,line);
      a.F1i1_clock = atoi(line.c_str());
  //****** EVALUATE SIGNALS IN FOLLOWER SET ELEMENT 1***
  // firing conditions for actors F1, F2, F3 are set
         a.F1IS1 = true; // input event arrives
         a.F2i6 = a.F1i1; a.F2IS6 = true; a.M2i10 = a.F1i1;
         a.M2IS10 = true; a.F3i11 = a.F1i1; a.F3IS11 = true;
         }
      else// *** No event occurred in master trigger signal ***
         {a.F1IS1 = false; a.F2IS6 = false;
          a.M2IS10 = false;a.F3IS11 = false;
         }

 //*******SECTION 2*******// Schedule and fire actors based on inputs
    if (a.F1IS1 == true)
       { a.F1o1 = a.fc1less (a.F1i1);
         a.F1OS1 = true;   }
       else   { a.F1OS1 = false;  }
    if(a.F1OS1==true)
      { a.S1i3=a.F1o1;a.S1IS3=true; }
      else {a.S1IS3=false;}
    a.Sampler(a.S1i2,a.S1i3,a.S1o2,a.S1IS2,a.S1IS3,a.S1OS2);
                ..     // Firing conditions and computations of other actors
                ..
    a.Send_Output_To_File(iteration);
    iteration++;
 }// end while loop
}
```

absolute_pib_syn_func.cpp

```
//************Contains all functions and standard actor definitions****
//Custom Actor F1
bool Synth::fc1less (int a)
 {if (a <0)
       return true;
  else
       return false;
 }
 .. OTHER ACTORS

//standard boolean Merge actor
void Synth::Merge(bool a,bool b,bool& c,bool d,bool e,bool& f)
 { if(d)
      {c = a;
 ...
   else
      {f = false;}
}
```

Figure 7.7: Generated code for `Absolute`

Chapter 8

Faster synthesis from MRICDF specifications using Actor elimination Technique

EmCodeSyn was built to enable the design and debug of MRICDF networks and to perform sequential software synthesis from them. The Boolean theory approach to synthesis was implemented using a prime implicate technique of identifying the master trigger signal and follower set. Prime implicate computation is central to the synthesis methodology and is the most time consuming part. 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.

8.1 Restrictions for preservation of signal epoch

Any optimization technique for faster follower set computation should not yield different follower set results. One optimization could be simplification of an 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 8.1.1 (Master Trigger Equivalence)**.** *Consider 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 an optimization technique. These 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 ($\widehat{x_i} = \widehat{x_j}$).*

According to **Thm. 6.3.1**, in each instant of an MRICDF network, a master trigger signal will have an event occurrence. So there is a close relation between a master trigger and instant set of a network. We now claim that to prove Master Trigger Equivalence, it is both necessary and sufficient to check if the instant sets of the two MRICDF networks are the same.

Theorem 8.1.1 (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. 8.1.1**, 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 Υ_1 and if any optimization results in a new MRICDF model M_2 with master trigger x_2 and an instant set Υ_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 (**Defn. 6.3.7**), 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 Υ_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, Υ_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 $\widehat{x_2} = \widehat{x_1}$. Hence, proved that two MRICDF models are Master Trigger equivalent, if they have the same instant set.

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, $\widehat{x_1}$ has changed to a new epoch, say $\widehat{x_2}$ which fulfills the master trigger condition for the new model M_2. The new set of abstract instants Υ_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 8.1.2 (Preservation of Prime implicate epoch)**.** *The elimination of any signal from an MRICDF network M_1 having prime implicate epoch $\widehat{p_1}$ results in a new MRICDF network M_2 with a prime implicate epoch $\widehat{p_2}$. 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. 8.1.1** 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. The epochs of all prime implicate signals are the same, including the master trigger signal. So, if the master trigger signal epochs are the same for two MTE MRICDF networks, then their prime implicate epochs are also the same.

8.2 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, Σ), where V is the set of variables and Σ 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 Σ_{elim} through AET a reduced MRICDF network (V', Σ') 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 $V = V_I \cup V_{oth}$, where V_I is the set of input variables and V_{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 $v_i = v_j$ becomes a part of the eliminated Boolean equation set Σ_{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}. Each elimination rule shows the transformation of an MRICDF network with eliminated epoch equations $(V, \Sigma, \Sigma_{elim})$ after each elimination rule is applied. It shows the variables eliminated from the MRICDF network, the signals replaced during AET, and the epoch equations that are eliminated from the MRICDF network.

$$\frac{V, \Sigma \cup \{y = x\}, \Sigma_{elim}}{V - \{x, y\}, \Sigma\{\frac{i_1}{x}, \frac{i_1}{y}\}, \Sigma_{elim} \cup \{x = y\}} \quad B_{elim}$$

$$\frac{V, \Sigma \cup \{z = f(\vec{x})\}, \Sigma_{elim}}{V - \{\Pi_i(\vec{x}), z\}, \Sigma\{\frac{i_1}{\Pi_i(\vec{x})}, \frac{i_1}{z}\}, \Sigma_{elim} \cup \{\Pi_i(\vec{x}) = z\}} \quad F_{elim}$$

$$\frac{V, \Sigma \cup \{z = x \cap [y]\}, \Sigma_{elim}}{V - \{x, y, z\}, \Sigma\{\frac{i_1}{x}, \frac{i_1}{y}, \frac{i_1}{z}\}, \Sigma_{elim} \cup \{z = x \cap [y]\}} \quad S_{elim}$$

$$\frac{V, \Sigma \cup \{z = x \cup x\}, \Sigma_{elim}}{V - \{x, z\}, \Sigma\{\frac{i_1}{x}, \frac{i_1}{z}\}, \Sigma_{elim} \cup \{z = x \cup x\}} \quad M_{elim}$$

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. 8.1(a) shows the application of AET on `Buffer` actor. An MRICDF network (V, Σ) is shown, consisting of two `Function` actors with a `Buffer` actor in between. It can be represented as $V = V_R \cup V_B$ and $\Sigma = \Sigma_R \cup \Sigma_P \cup \Sigma_B$, where V_B represents the variables of `Buffer` actor and V_R is the set of all other variables in the network. Let Σ_B represent epoch equations with only `Buffer` actor variables and Σ_P represent equations with input and output port connections of `Buffer`. Then Σ_R represents the equations represented by rest of the MRICDF network.
$\Sigma = \Sigma_R \cup \Sigma_P \cup \Sigma_B$
$\Sigma = \Sigma_R \cup \{i_1 = x\} \cup \{y = o_1\} \cup \{x = y\}$

Going by the elimination rule B_{elim}, the epoch equation $\{x = y\}$ is eliminated and thus Σ_B becomes a part of Σ_{elim}. The port connections x, y are replaced with i_1, making equations $\{i_1 = x\}$ 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', Σ') is MTE to the original network (V, Σ), where $V = V' \cup \{x, y\}$ and $\Sigma = \Sigma' \cup \Sigma_{elim}$.

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. 8.1(b). The input-output signals $x, y,$ and z will be eliminated under AET rules.

The MRICDF network in Fig. 8.1(b) is represented by (V, Σ), where $\Sigma = \Sigma_1 \cup \Sigma_P \cup \Sigma_F$. Here Σ_F represents the Boolean equations of the actor F_1 that will be eliminated. $\Sigma = \Sigma_R \cup \Sigma_P \cup \Sigma_F$
$\Sigma = \Sigma_R \cup \{i_1 = x\} \cup \{i_2 = y\} \cup \{z = o_1\} \cup \{x = z\} \cup \{y = z\}$
After elimination, Σ_F is a part of Σ_{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', Σ'), $V = V' \cup \{x, y, z\}$ and $\Sigma = \Sigma' \cup \Sigma_{elim}$, 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.

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

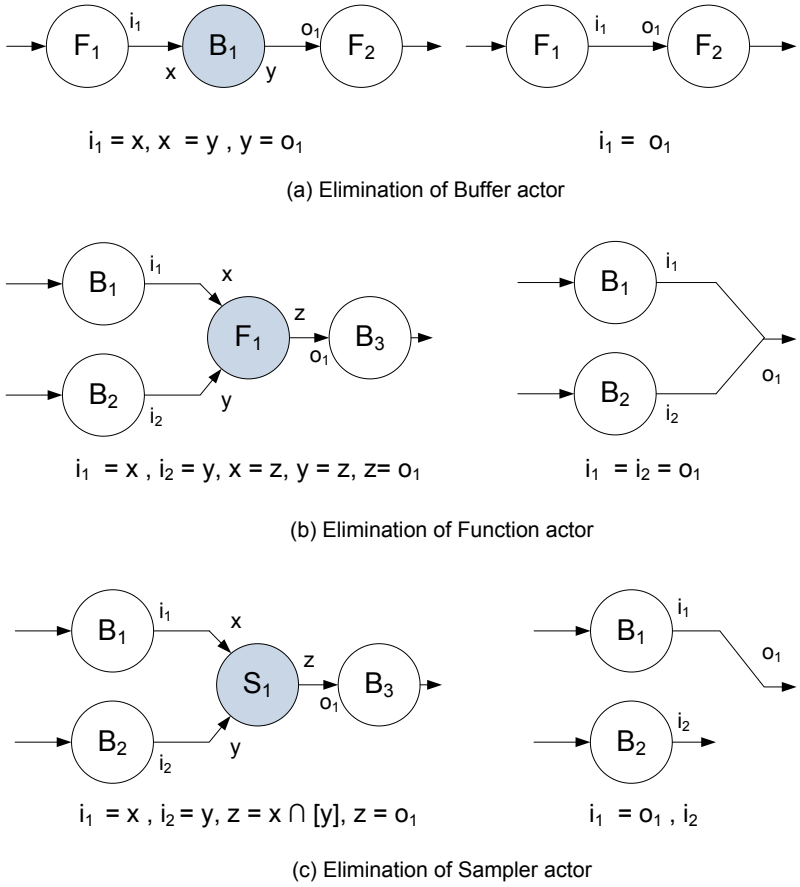


Figure 8.1: Actor elimination on MRICDF primitive actors

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. 8.1(c) a `Sampler` actor is shown to be connected to three `Buffer` actors. The MRICDF network is represented by Boolean equations Σ and the `Sampler` actor Boolean equations are represented by Σ_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_S, Σ_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 MRCIDF network is an input to S_1 actor (say x), the epoch of S_1 actor variables will be of the form $\widehat{[y]} \subseteq \widehat{y} \subseteq \widehat{x}$ and $\widehat{z} \subseteq \widehat{x}$. So the `Sampler` epoch equation reduces from $\widehat{z} = \widehat{x} \cap \widehat{[y]}$ to $\widehat{z} = \widehat{[y]}$. 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 $\widehat{z} = \widehat{x} = \widehat{[y]}$. 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 m), we know $\widehat{x}, \widehat{y}, \widehat{z} \subseteq \widehat{m}$. Replacing any of the `Sampler` internal signals $x, y,$ or 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.

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.

8.3 Implications of actor elimination on follower set generation

Now let us discuss where AET is applied with respect to the follower set computation. Consider the example given in Fig. 8.2(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. 8.2(b) is evaluated and no master trigger is found. External master trigger signals $x0\_S1.i.1, x1\_S1.i.2, x2\_S2.i.3, x3\_S2.i.4$ are applied to provide clock information for each input. On true occurrences of these clock signals values are read from input ports $S1.i.1, S1.i.2, S2.i.3, S2.i.4$. $b_{S1.i.1} + b_{S1.i.2} + b_{S2.i.3} + b_{S2.i.4}$ is among the prime implicates of the newly created Boolean equations. The `Sampler` actors `S1,S2` can only be fired on true values at ports $S1.i.2$ and $S2.i.4$. Hence the signals $[S1.i.2], [S2.i.4]$ are part of the third FS element along with signals that can be computed as the actors `S1,S2` fires. Fig. 8.2(c) shows the actors in shaded portion, which can be evaluated for scheduling with the current epoch information. On true values at input port $S3.i.9$, the actors `S3,M1` will be available for scheduling.

We have already discussed that irrespective of AET rules, input signals connected to any MRICDF actor cannot be eliminated. So any primitive actor with an input to the environment is not considered during elimination. Also, in certain oversampling situations where an internal signal is

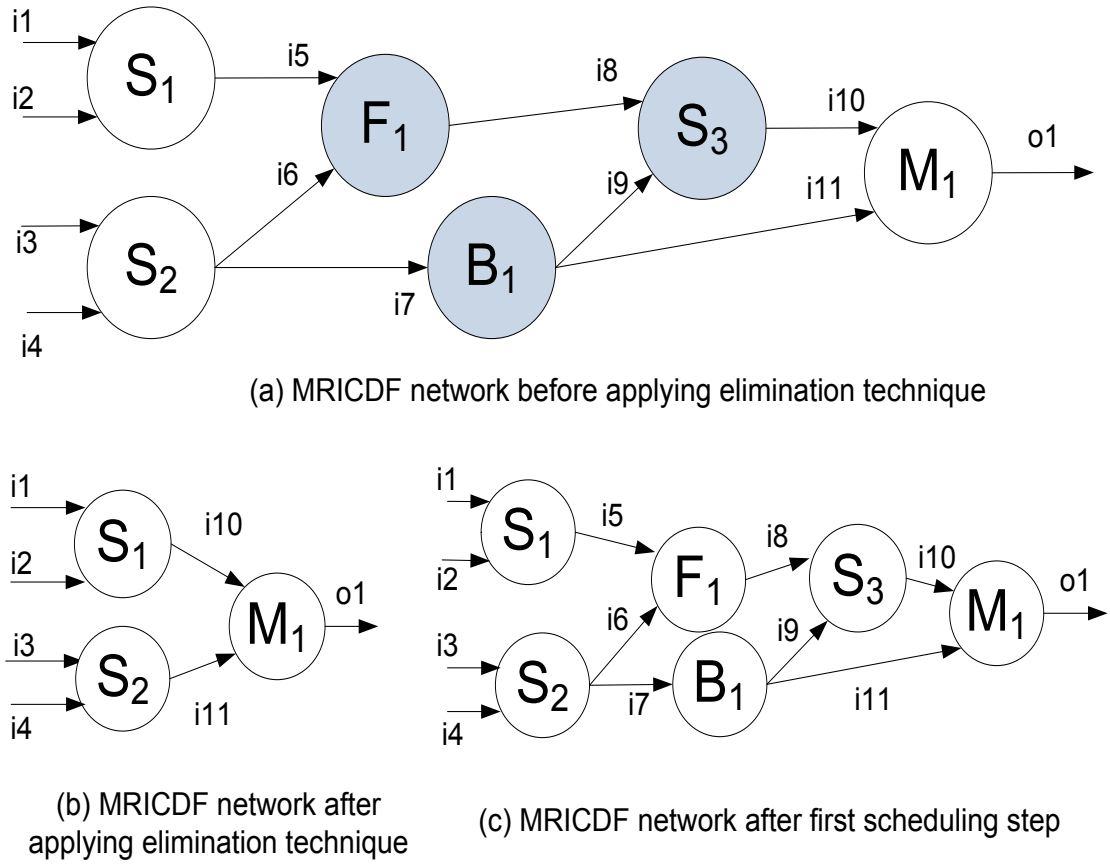


Figure 8.2: Elimination steps and scheduling for an MRICDF network

the master trigger, actors in a loop creating the master trigger signal cannot be eliminated. For instance, if elimination of actors result in a clock equation $\widehat{a} = \widehat{b} \cup \widehat{a}$, the clock of b becomes irrelevant. If b is an external input, this equation assumes the instants of b is a subset of instants of a, which need not be true in the original MRICDF network. So while eliminating `Sampler` actors connected to `Merge` actors, we look for possible causal loops containing them which can create wrong clock equations.

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.6.3.2**) 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 8.3.1 (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. 8.1.1** 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 8.3.2 (Follower set preservation)**.** *For any MRICDF network (say $M1$) with follower set FS_{M1} and its optimized version $M2$ with follower set FS_{M2} obtained on applying AET , the epoch of follower set elements FS_{M1} and FS_{M2} are the same.*

. **Proof:** Preservation of master trigger (**Thm. 8.1.1**) proved that for two MTE MRICDF networks, the instant set remains the same and it preserves the master trigger signal epoch. **Cor. 8.1.2** has shown that preserving prime implicate epoch is same as preserving instant set of the original MRICDF network. By **Defn. 6.3.8**, 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. ■

8.3.1 Case studies on application of Actor Elimination Technique

The optimizations for EmCodeSyn were targeted towards prime implicate computation. Interfaced with an earlier version of prime implicate generator, the software synthesis time was around 420 seconds. The new PI generator customized for EmCodeSyn brought down code generation time for `Absolute` to 3 seconds. 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. 8.3 shows the application of AET on the MRICDF network `Absolute` which was discussed in Chapter 7. Here three `Sampler` actors will be eliminated and both `Merge` actors will be retained as per elimination rules. The `Function` actors will be retained as they are connected to inputs from the environment. 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.

To demonstrate how AET is applied on larger MRICDF models, we consider the case study of a watchdog timer. The SIGNAL code of this watchdog timer is shown in Listing 8.1 [64]. The watchdog timer consists of a clock signal *tick* and two other inputs *req* and *finish*. *req* indicates the arrival of a new timing request and *finish* indicates an interruption to the count down process. For every request a constant value *delay* is loaded and a decrement operation is performed. *cnt* shows the decremented value and when it reaches zero, an output signal is triggered at the port *alarm*.

Listing 8.1: SIGNAL code for watchdog timer [64]

```
1  process Watchdog =  { integer delay; }
2    ( ? integer req; event finish, tick;
3      ! integer alarm; )
4   (| hour ^= tick
5    | hour := (hour$ init 0) + 1
```

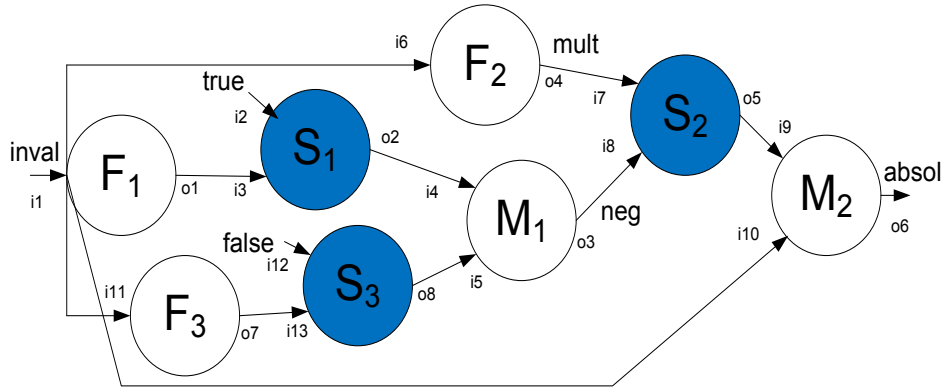


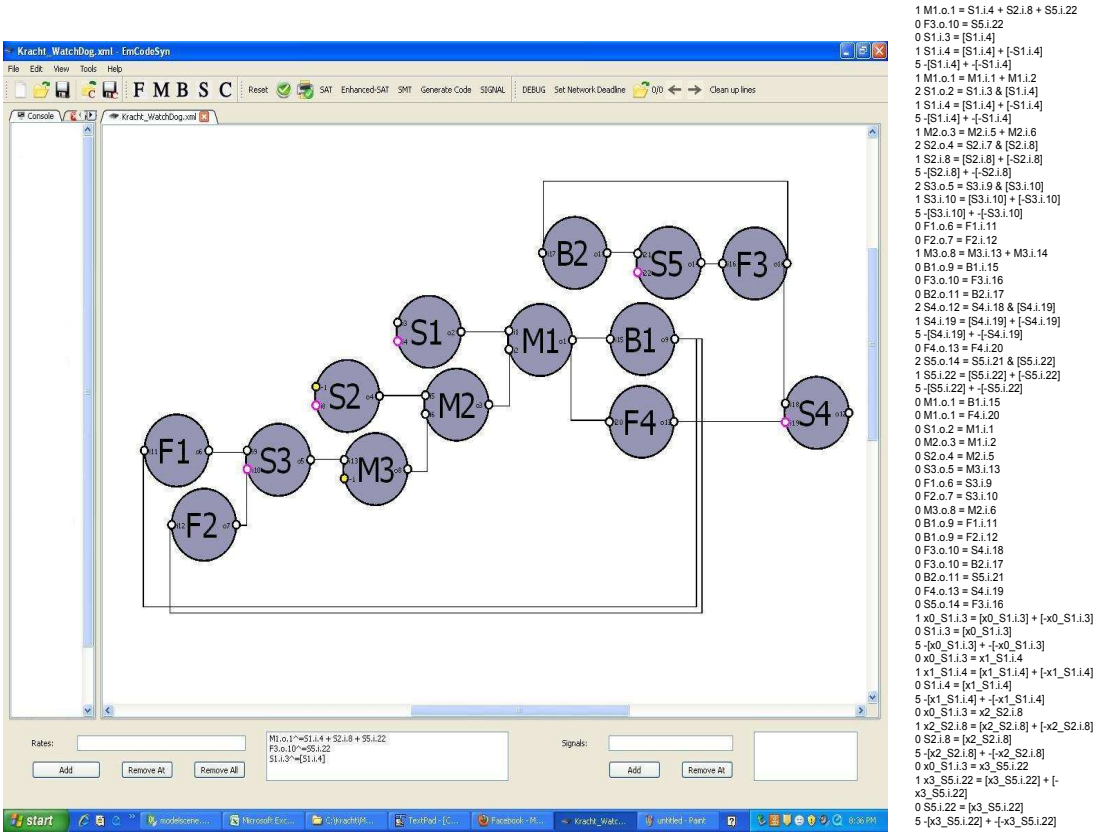
Figure 8.3: Actor elimination technique applied on the example `Absolute`

```
6     | cnt ^= tick ^+ req ^+ finish
7     | prev_cnt := cnt$ init (-1)
8     | cnt := (delay when ^req)
9     default (-1 when finish)
10    default ((prev_cnt - 1) when (prev_cnt >= 0))
11    default -1
12    | alarm := hour when (cnt = 0)  |)
13      where  integer hour, cnt, prev_cnt;
14  end;
```

The MRICDF visual representation of the watchdog timer is shown in Figure 8.4 (a). There are 14 primitive MRICDF actors in the network, with clock constraints given at the bottom. The 60 epoch equations that represent the watchdog timer actors and constraints are shown in the right hand side. On application of AET rules, the number of actors is reduced to 8 and the epoch equations are reduced to 48 as shown in Figure 8.4 (b). The `Merge` actors are completely retained as per AET rules. Actors `S1,S2,S4` are retained since they are connected to input-output ports, while `S3,S5` were not eliminated due to the presence of a loop. The master trigger signals were observed to be of the same epoch, with and without actor elimination. All master trigger signals of MRICDF network in Figure 8.4 (b) were found among the master trigger signals of its original MRICDF network in Figure 8.4 (a). Repeated PI generation process builds the complete follower set for the network, and was observed to have the same epoch [87].

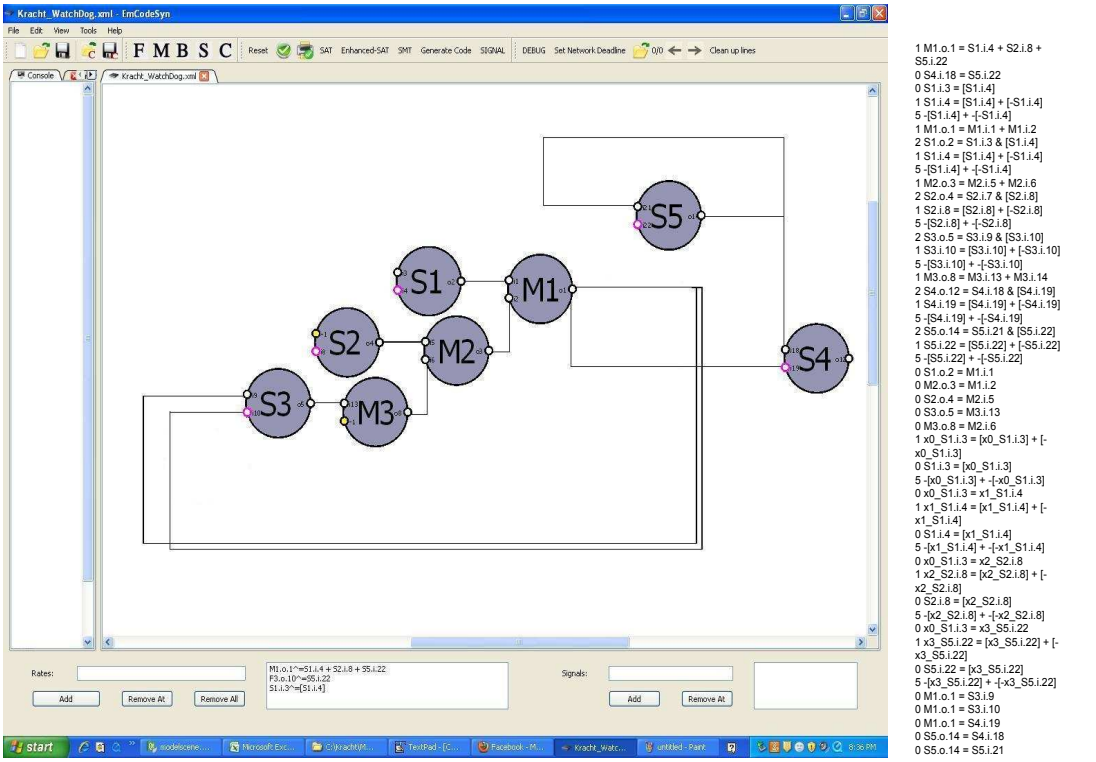
8.4 Impact of Actor Elimination Technique on code synthesis

Let us discuss how actor elimination technique is integrated into the PI based design methodology using the flow diagram in Fig. 8.5. The MRICDF network is converted into epoch equations and AET is applied on each internal actor. The epoch equations are converted to Boolean equations along with a constraint to avoid trivial solutions. This is sent to PI generator to find a master trigger. In the absence of a master trigger, exogenous information is requested to enforce deterministic




```
1 M1.o.1 = S1.i.4 + S2.i.8 + S5.i.22
0 F3.o.10 = S5.i.22
0 S1.i.3 = [S1.i.4]
1 S1.i.4 = [S1.i.4] + [-S1.i.4]
5 -[S1.i.4] + -[-S1.i.4]
1 M1.o.1 = M1.i.1 + M1.i.2
2 S1.o.2 = S1.i.3 & [S1.i.4]
1 S1.i.4 = [S1.i.4] + [-S1.i.4]
5 -[S1.i.4] + -[-S1.i.4]
1 M2.o.3 = M2.i.5 + M2.i.6
2 S2.o.4 = S2.i.7 & [S2.i.8]
1 S2.i.8 = [S2.i.8] + [-S2.i.8]
5 -[S2.i.8] + -[-S2.i.8]
2 S3.o.5 = S3.i.9 & [S3.i.10]
1 S3.i.10 = [S3.i.10] + [-S3.i.10]
5 -[S3.i.10] + -[-S3.i.10]
0 F1.o.6 = F1.i.11
0 F2.o.7 = F2.i.12
1 M3.o.8 = M3.i.13 + M3.i.14
0 B1.o.9 = B1.i.15
0 F3.o.10 = F3.i.16
0 B2.o.11 = B2.i.17
2 S4.o.12 = S4.i.18 & [S4.i.19]
1 S4.i.19 = [S4.i.19] + [-S4.i.19]
5 -[S4.i.19] + -[-S4.i.19]
0 F4.o.13 = F4.i.20
2 S5.o.14 = S5.i.21 & [S5.i.22]
1 S5.i.22 = [S5.i.22] + [-S5.i.22]
5 -[S5.i.22] + -[-S5.i.22]
0 M1.o.1 = B1.i.15
0 M1.o.1 = F4.i.20
0 S1.o.2 = M1.i.1
0 M2.o.3 = M1.i.2
0 S2.o.4 = M2.i.5
0 S3.o.5 = M3.i.13
0 F1.o.6 = S3.i.9
0 F2.o.7 = S3.i.10
0 M3.o.8 = M2.i.6
0 B1.o.9 = F1.i.11
0 B1.o.9 = F2.i.12
0 F3.o.10 = S4.i.18
0 F3.o.10 = B2.i.17
0 B2.o.11 = S5.i.21
0 F4.o.13 = S4.i.19
0 S5.o.14 = F3.i.16
1 x0_S1.i.3 = [x0_S1.i.3] + [-x0_S1.i.3]
0 S1.i.3 = [x0_S1.i.3]
5 -[x0_S1.i.3] + -[-x0_S1.i.3]
0 x0_S1.i.3 = x1_S1.i.4
1 x1_S1.i.4 = [x1_S1.i.4] + [-x1_S1.i.4]
0 S1.i.4 = [x1_S1.i.4]
5 -[x1_S1.i.4] + -[-x1_S1.i.4]
0 x0_S1.i.3 = x2_S2.i.8
1 x2_S2.i.8 = [x2_S2.i.8] + [-x2_S2.i.8]
0 S2.i.8 = [x2_S2.i.8]
5 -[x2_S2.i.8] + -[-x2_S2.i.8]
0 x0_S1.i.3 = x3_S5.i.22
1 x3_S5.i.22 = [x3_S5.i.22] + [-
x3_S5.i.22]
0 S5.i.22 = [x3_S5.i.22]
5 -[x3_S5.i.22] + -[-x3_S5.i.22]
```

(a) Watchdog Timer example before actor elimination




```
1 M1.o.1 = S1.i.4 + S2.i.8 +
S5.i.22
0 S4.i.18 = S5.i.22
0 S1.i.3 = [S1.i.4]
1 S1.i.4 = [S1.i.4] + [-S1.i.4]
5 -[S1.i.4] + -[-S1.i.4]
1 M1.o.1 = M1.i.1 + M1.i.2
2 S1.o.2 = S1.i.3 & [S1.i.4]
1 S1.i.4 = [S1.i.4] + [-S1.i.4]
5 -[S1.i.4] + -[-S1.i.4]
1 M2.o.3 = M2.i.5 + M2.i.6
2 S2.o.4 = S2.i.7 & [S2.i.8]
1 S2.i.8 = [S2.i.8] + [-S2.i.8]
5 -[S2.i.8] + -[-S2.i.8]
2 S3.o.5 = S3.i.9 & [S3.i.10]
1 S3.i.10 = [S3.i.10] + [-S3.i.10]
5 -[S3.i.10] + -[-S3.i.10]
1 M3.o.8 = M3.i.13 + M3.i.14
2 S4.o.12 = S4.i.18 & [S4.i.19]
1 S4.i.19 = [S4.i.19] + [-S4.i.19]
5 -[S4.i.19] + -[-S4.i.19]
2 S5.o.14 = S5.i.21 & [S5.i.22]
1 S5.i.22 = [S5.i.22] + [-S5.i.22]
5 -[S5.i.22] + -[-S5.i.22]
0 S1.o.2 = M1.i.1
0 M2.o.3 = M1.i.2
0 S2.o.4 = M2.i.5
0 S3.o.5 = M3.i.13
0 M3.o.8 = M2.i.6
1 x0_S1.i.3 = [x0_S1.i.3] + [-
x0_S1.i.3]
0 S1.i.3 = [x0_S1.i.3]
5 -[x0_S1.i.3] + -[-x0_S1.i.3]
0 x0_S1.i.3 = x1_S1.i.4
1 x1_S1.i.4 = [x1_S1.i.4] + [-
x1_S1.i.4]
0 S1.i.4 = [x1_S1.i.4]
5 -[x1_S1.i.4] + -[-x1_S1.i.4]
0 x0_S1.i.3 = x2_S2.i.8
1 x2_S2.i.8 = [x2_S2.i.8] + [-
x2_S2.i.8]
0 S2.i.8 = [x2_S2.i.8]
5 -[x2_S2.i.8] + -[-x2_S2.i.8]
0 x0_S1.i.3 = x3_S5.i.22
1 x3_S5.i.22 = [x3_S5.i.22] + [-
x3_S5.i.22]
0 S5.i.22 = [x3_S5.i.22]
5 -[x3_S5.i.22] + -[-x3_S5.i.22]
0 M1.o.1 = S3.i.9
0 M1.o.1 = S3.i.10
0 M1.o.1 = S4.i.19
0 S5.o.14 = S4.i.18
0 S5.o.14 = S5.i.21
```

(b) Watchdog Timer example after actor elimination

Figure 8.4: Actor Elimination Technique applied on Watchdog timer example

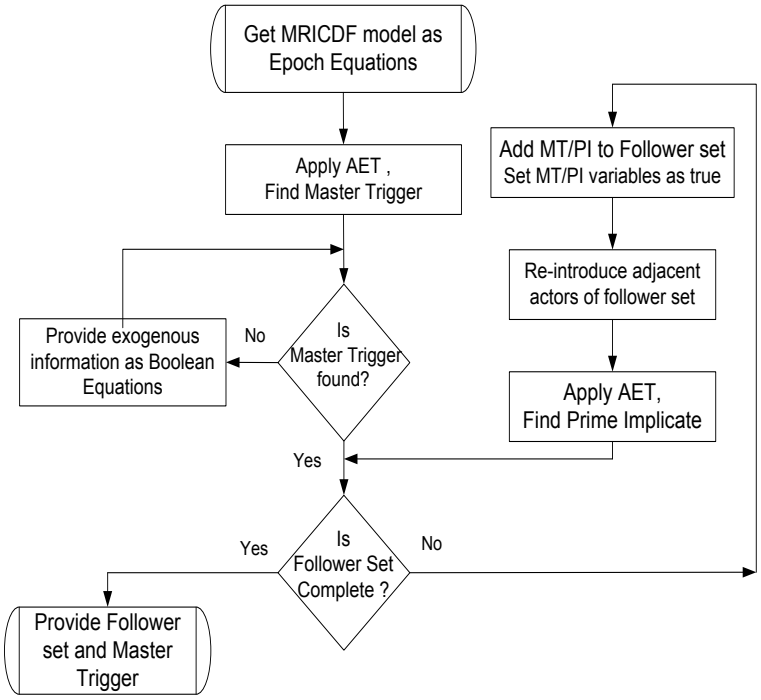


Figure 8.5: Generation of follower set using Prime Implicate method

sequential implementability or endochrony on the model. Once master trigger is identified, we check if the follower set is complete. This can happen if the model is a single primitive actor. If incomplete, we set the Boolean variable corresponding to the master trigger to true and generate a new set of Boolean constraints. Now actors which are adjacent to signals in the current follower set and were removed during the AET step are re-introduced. The updated system of equations are sent to the PI generator to get the next set of elements of the follower set. Once this iterative process is complete, the follower set will contain every signal in the MRICDF network. Once the follower set is established, code generation phase can commence. The importance of AET is highlighted by the fact that the time consuming prime implicate computation is a repetitive step for follower set generation. In each prime implicate computation step, AET will bring down waiting time with negligible overhead.

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 hand-shaking protocol module. More information about these examples are summarized in a technical report with references [87]. The impact of AET on synthesis time can be understood by analyzing the numbers in Table 8.1. 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 master trigger detection time in seconds is given in columns 5 and 6. These numbers are directly

MRICDF network	**♯ of actors**	**♯ of Boolean eq.**		**MT computation (sec)**		**Synthesis (sec)**	
		Original	AET	Original	AET	Original	AET
Height-supervisor	5	14	14	0.359	0.36	1.2	1.2
Absolute	8	24	11	0.906	0.906	3	2
Factorial	8	25	15	0.328	0.329	1.4	1.3
Resettable counter	8	24	18	0.922	0.563	2.5	1.8
pEHBH	14	52	49	0.75	0.719	3.5	3.3
Watchdog timer	14	60	48	16.313	5.375	125.4	53.7
Producer-consumer	15	50	44	21.343	16.328	46.7	34.8
Flight Warning Syst.	17	48	26	8.109	1.032	329.5	4.7

Table 8.1: Impact of AET on software synthesis time in EmCodeSyn

related to the number and type of Boolean equations being operated on by the prime implicate generator. The total synthesis time for generating C files in seconds is given in columns 7 and 8. 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 [61] 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.

Chapter 9

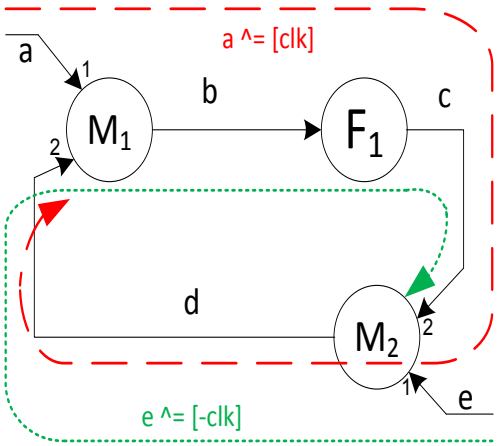
Improved Casual loop detection and faster synthesis using SMT solvers

Instantaneous or causal loops, i.e., cyclic data dependency within an instant, have been a problem in all languages in the synchronous programming paradigm. Various solutions to the problem of detecting causal loops and non-realizable apparent causal loops, referred to as *false causal loop*, have appeared in the literature of Esterel [133], Lustre [75], SIGNAL [71], and hardware description languages. In general, the problem is undecidable because determining if a loop is realizable may be dependent on arbitrary data types, and arbitrary functions on such data. As a result, approximations have been proposed. To err on the side of the caution, due to false positives, a lot of correct programs are rejected by compilers. In the early days of synchronous programming, syntactic checks were used, leading to too many false causal loops. Since then, semantic analysis have been incorporated in many of the compilers. Unfortunately, in the polychronous programming literature, not a lot of attention has been paid to this problem beyond 1) syntactic detection rejecting all buffer-free loops as implemented in the Polychrony compiler [106] and 2) causal loop detection using nullity check of clock intersection in SIGNAL programs [34]. With the advent of SAT Modulo theory checkers, these approximations dealing with Boolean data types can be vastly improved. Also, the causal loop detection techniques can be extended to analyze nuances of polychronous calculus and thus detect falsity of apparent causal loops. We have covered the existing works relating to causality analysis of polychronous and synchronous specification in Chapter 3. In this chapter, we revisit the problem of false causal loops in polychronous specifications, and show how we solve this problem in our MRICDF framework using SMT solvers and model checking tools. We also investigate if our causality analysis improvements can be utilized to provide faster synthesis.

9.1 Motivational examples for better causality analysis

We first illustrate the vulnerabilities of existing causality detection tools and motivate why verification tools such as SMT solvers and model checkers are required for better causality analysis. We introduce specifications with possible causal loops and discuss the capability of Polychrony and EmCodeSyn in detecting them.

Example 9.1.1 (Causal loop detection)**.** *A polychronous MRICDF network* `Spec1` *with its equivalent SIGNAL program are shown in Figure 9.1. There are two* `Merge` *MRICDF actors* M_1 *and* M_2 *which merge input streams with a higher priority given to first input port on occurrences of simultaneous events.*




```
process Spec1 =
( ? boolean a, e, clk;
  ! boolean b; )
( | b := a default d     %M1%
  | c := b+1             % F1%
  | d := e default c     % M2%
  | a ^= when clk
  | e ^= when (not(clk))
  |)
where boolean c, d;
end;
```

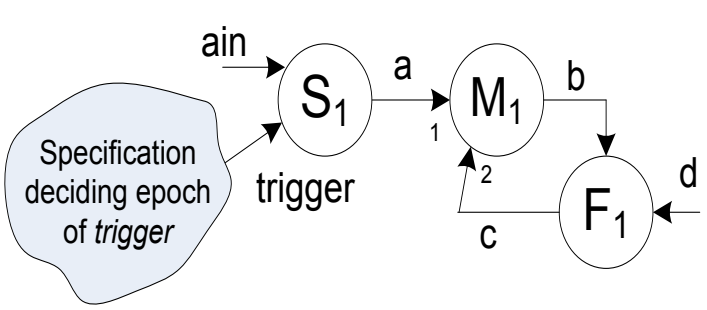
Figure 9.1: Spec1: causal loop specification in MRICDF (left) and SIGNAL (right)

`Merge` *actors (here* M_1 *and* M_2*) give priority to port 1 over port 2. An external signal* `clk` *decides the flow of data in the specification. When* `clk` *is present and true (shown as* `[clk]`*), there are events on* `a` *connected to* M_1 *and not on* `e`*. Consequently,* M_1 *ignores inputs on port 2, which means the data does not flow through the entire loop as shown by the red dashed lines. When* `clk` *is present and* $false$ *(shown as* `[- clk]`*), input signals are present on signal* `e` *connected to* M_2 *and not on* `a`*. Now, the flow of data is shown by dotted green lines. In both cases, the edges of the data flow network loop, i.e., signals* `b, c` *and* `d` *are active. But, there is no causal loop* $b \rightarrow c \rightarrow d \rightarrow b$*. Also, there is no possible input for the specification which can lead to a causal loop.*

Polychrony compiler and EmCodeSyn cannot synthesize sequential code for the program `Spec1` in Figure 9.1. A dependency cycle is reported between variables `b`, `c` and `d`. But, we learned how the polychronous nature of the specification ensured a data dependent loop does not arise. The clock tree representation of Polychrony and the Boolean theory representation of EmCodeSyn are insufficient to determine implementability of such specifications.

Example 9.1.2 (Causal loop formed by clock and data constraints)**.** *A polychronous MRICDF network* `Spec2` *with its equivalent SIGNAL program are shown in Figure 9.2. There is a* `Merge` *actor* M_1*, a* `Sampler` *actor* S_1 *and a* `Function` *actor* F_1 *in the MRICDF network. The* `Sampler`

actor S_1 *passes the value of any event at* `ain` *onto* `a` *when the Boolean input* `trigger` *has a* $true$ *valued event.* `Function` *actor* F_1 *computes the sum of* `b` *and* `d`*. Two possible constraints (C1,C2) are shown with C1 included as a part of SIGNAL code.*




```
process Spec2 =
( ? boolean trigger;
    integer ain, d;
  ! integer b; )
(| a := ain when trigger  % S1%
 | b := a default c       %M1%
 | c := b + d             %F1%
 | ain  ^= d ^=  trigger |)%C1%
 where integer a, c;
 end;
```

Figure 9.2: Spec2: Evaluating causal loop conditions with clock and data constraints

In the SIGNAL program `Spec2`*, due to the clock constraint C1, a continuous stream of events is guaranteed at signals* `ain, d` *and* `trigger`*. When* `trigger` *signal is present with a* $false$ *value, there is no event at* `a`*. Here, a causal loop (*`b` $\rightarrow$ `c` $\rightarrow$ `b`*) may be formed. In order to avoid this situation, clock constraint C2 can be applied guaranteeing always* $true$ *values at* `trigger` *(represented as* `[trigger]`*). Since* `Merge` *actor has a higher priority for port 1, events at port 2 are ignored and thus a causal loop is avoided.*

Polychrony and EmCodeSyn will reject `Spec2` with constraint `C1`, accept `Spec2` with constraint `C2` and rightly so. If `trigger` is not an external input, rather a data computation `D1` requiring Integer arithmetic (say `trigger :=` $false$ `when (ain >5 and ain <2) default` $true$), both tools will reject the specification. Here, `trigger` can be $false$ only when `ain` is greater than 5 and less than 2, which is impossible. But the intermediate structures of Polychrony and EmCodeSyn are not powerful enough to capture and evaluate these conditions. In fact, if code generation is forced on Polychrony, a causal loop (`b` $\rightarrow$ `c` $\rightarrow$ `b`) is warned. In summary, there is a need to move onto more expressive and powerful structures which expand the set of implementable polychronous programs using a software tool.

The prime implicate technique of implementing Boolean theory approach to sequential synthesis was found to be time consuming in Chapter 8. The prime implicate generation time for master trigger identification is long, making EmCodeSyn impractical for larger designs. Enhancements such as Actor Elimination Technique simplify the Boolean equations which PI generator operates on. Any further improvement would require better prime implicate generation algorithms or alternatives to PI technique of implementing Boolean theory based sequential synthesis. In a recent work, verification tools were shown to be efficient in software synthesis for generic computing algorithms [128]. Program guards are evaluated as verification problems to determine program execution. One popular verification tool, the SMT solver [57], is capable of evaluating theories

of various data structures as motivated by Example 9.1.2. Recent developments in SAT and SMT solvers [36] have made their use in synthesis practical. In this chapter, we show how the synthesis conditions of MRICDF networks can be posed as satisfiability problems that can be solved using Satisfiability modulo theories. Thus, we show how the time consuming prime implicate based implementation cane be replaced with an SMT implementation of the Boolean theory approach to synthesis.

9.2 Causal loop detection on polychronous specifications

Among the tools adopting a polychronous MoC, Polychrony and EmCodeSyn analyze clock (or epoch) relations in specifications to determine if a causal loop exists. We differentiate the kinds of causal loops in polychronous specifications and explain how we improve the existing loop detection techniques. First we review the definition of apparent causal loop from Chapter 6.

Definition 9.2.1 (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 an instant where all signals in the topological loop have an event. $S \in \Upsilon, \forall_{i=1}^{n}, \exists e \in E(x_i) \wedge e \rhd S$.*

`Spec1` is a program rejected after causality analysis in Polychrony since the signals `b, c` and `d` form an apparent causal loop. Current versions of EmCodeSyn will do the same by detecting a possible logical instant ($b_b \wedge b_c \wedge b_d = true$) resulting in an apparent causal loop. Due to the nature of primitives such as `Merge` and `default` where a lower priority input might get ignored, a signal with an event does not mean there exists a causal loop with cyclic data dependency. We have demonstrated this using our example `Spec1`.

Definition 9.2.1 (False Causal Loop)**.** *Consider an apparent causal loop with signals $x_1, x_2, .., x_n$ having event sets $E(x_1), E(x_2), .., E(x_n)$. An apparent causal loop is considered to be a false causal loop, if and only if, for no instant in Υ, there exists a data dependency between events $e_n \in E(x_n)$ with the events $e_1 \in E(x_1)$ and for signals $x_1, x_2, .., x_{n-1}$ there exists a data dependency between events $e_i \in E(x_i)$ with the events $e_{i+1} \in E(x_{i+1})$.*

Definition 9.2.2 (True Causal Loop)**.** *A true causal loop is an apparent causal loop which is not a false causal loop.*

If the edges of a loop have events that never happen on the same logical instants, it is a false causal loop. A reason for incorrect rejection of programs due to false causal loops is the weakness of intermediate structures used for analysis. Clock analysis alone cannot entirely guarantee presence of an event, due to dependency on values of functional computation units as demonstrated by our example `Spec2`. Neither clock calculus of Polychrony nor Boolean epoch analysis of EmCodeSyn

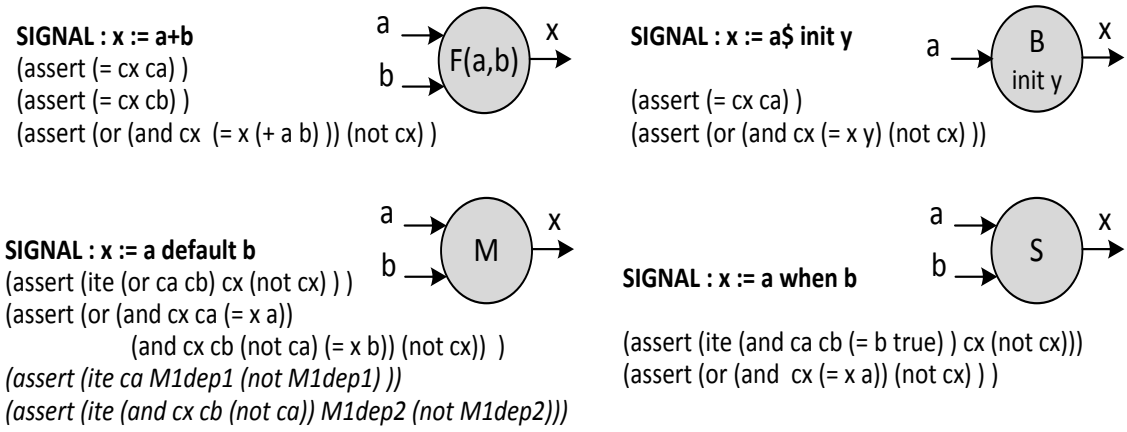


Figure 9.3: SMT equations for MRICDF/SIGNAL primitives

can fully capture the actual data dependencies in their intermediate structures. We believe satisfiability modulo theories are capable of accommodating these requirements. Hence, they are chosen to distinguish between true and false causal loops.

9.2.1 Detection of true causal loops using SMT solvers

Satisfiability (SAT) of a Boolean formula implies there exists a solution to the encoded problem. Satisfiability Modulo Theory (SMT) extends SAT by checking satisfiability of formula over multiple theories such as Boolean, Integer and so on [57]. Our chosen SMT solver Yices version 1.0.29 has a representation format which uses assertions with other common operators. The SMT representation of MRICDF/SIGNAL primitives is summarized in Figure 9.3.

A signal `x` is encoded by an epoch variable cx of Boolean type and a data variable x of same type as the signal. The epoch equations of `Function` and `Buffer` actors are shown as direct equations of input and output epoch variables in Figure 9.3. For `Sampler` and `Merge`, epoch equations are represented in a conditional *if-then-else* statement. Let us discuss `Merge` actor in detail, where the epoch equation is of the form '**if** *any of the input epoch variables are true*, **then** *output epoch variable is true*, **else** *output epoch is false*'. For data equations, the appropriate data dependency is chosen according to input firing conditions. In `Merge` data equation, the respective data dependencies are chosen according to the output variable value of cx along with the presence or absence of input epoch variables ca and cb. To make the analysis of cyclic dependencies easier, we put additional data dependency variables $M1dep1$ and $M1dep2$ to show the data dependency between input ports of the `Merge` actor with the output port. Note that only one among the data dependency variables can be true in a logical instant. This will help in determining flow of data in `Merge` actor along with favorable clock conditions in a loop.

Theorem 9.2.1 (True causal loop detection procedure)**.** *Given an MRICDF network M represented by the SMT formula set S_M and a set of causal loop candidates C represented by the SMT formula*

$S_C, \forall S_C^i \in S_C$, *if* $S_M, S_C^i \vDash true$, *then the MRICDF network has a true causal loop* S_C^i.

Proof sketch: For an SMT formula to be satisfiable, there exists at least one valid assignment to all the variables in the formula. The SMT formula set S_M is representative of any arbitrary instant $P \in \Upsilon$ of the MRICDF network M. The new SMT formula S_M, S_C^i is representative of all instants of MRICDF network where causal loop condition exists, denoted by Υ^i.
To prove that a given causal loop condition S_C^i is a **true causal loop**, we need to prove that any instant L where S_C^i holds, is in Υ. If $S_M, S_C^i \vDash true$, the SMT solver result is **satisfiable**. There exists at least one valid assignment to all the variables in the formula, or one instant of the MRICDF network with the causal loop. Hence it is a true causal loop ($\Upsilon^i \neq \phi$). If $S_M, S_C^i \vDash false$, the SMT solver result is **unsatisfiable**. This means there is no valid assignment to the variables in the formula. In other words, there does not exist any instant where the causal loop condition holds. Hence, S_C^i represents a **false causal loop** ($\Upsilon^i = \phi$).■

The SMT representation for `Spec1` in Figure 9.1 is given in Figure 9.4 (a). All signal definitions are of Boolean type and some are omitted in the figure. A *non-triviality condition* is added after the translation to remove the stuttering instant where no events can occur. Assertion 10 restricts the SMT equations to those instants where at least one signal has an event. Assertion 11 represents the causal loop condition under test. An UNSAT result is obtained from the SMT solver confirming the candidate to be a false causal loop.

The SMT representation for `Spec2` with the clock constraint $C1$ and data computation for `trigger` signal is shown in Figure 9.4 (b). Due to the clock constraint, a continuous stream of events is guaranteed at `ain`, `d` and `trigger`. The data computation constraint `D1 trigger :=` $false$ `when (ain >5 and ain <2) default` $true$ is implemented using three MRICDF actors F_2, S_2, and M_2. The apparent causal loop (`b` $\rightarrow$ `c` $\rightarrow$ `b`) is realizable only when a $false$ value is obtained at `trigger`. An UNSAT result is obtained verifying it to be false causal loop. When the data computation constraint is altered to `trigger :=` $false$ `when (ain >0 and ain <2) default` $true$, we did obtain a SAT result, reaffirming the possibility of a true causal loop.

Presence of a true causal loop does not mean the generated code will encounter the causal loop condition during execution. The realizability of causal loop condition using SMT technique does not check if the MRICDF network can arrive at the state where the causal loop conditions hold, from a given initial state. `Spec3`, a modified version of the MRICDF network `Spec2`, where clock constraint $C1$ is applied and the `trigger` signal is determined by another computation is shown in Figure 9.5 (a). Two incremental buffered counters `x` and `y` are shown with different initial values. Their increment operation is dependent on two signals having the same epoch, thereby equating the number of increments on them. The `trigger` signal is set to $false$, forming the true causal loop (`b` $\rightarrow$ `c` $\rightarrow$ `b`) only when `x` and `y` have the same value. With the initial conditions being different and the number of increments being the same, the true causal loop condition is not reachable. The SMT technique can identify the true causal loop, but cannot predict if the loop will ever occur in the course of execution of the generated code. Additional reachability analysis using model checkers is required for this purpose. The causal loop types and the set of tools that are capable of detecting each of them is shown in Figure 9.5 (b). Current versions of Polychrony

```
(set-evidence! true)
(define ca::bool)
(define a::bool)
(define cb::bool)
(define b::bool)
..
(define cclk::bool)
(define M1dep2::bool)
(define M2dep2::bool)

;;M1    b := a default d
1  (assert (ite (or ca cd) cb (not cb)))
2  (assert (or (and cb ca (= b a))(and cb cd(not ca)(= b d))
                (and (not cb)(not ca)(not cd))))

;;M2    d := e default c
3  (assert (ite (or ce cc) cd (not cd)))
4  (assert (or (and cd ce (= d e))(and cd cc(not ce)(= d c))
               (and (not cd)(not ce)(not cc))))

;;F1    c = b
5  (assert (= cc cb))
6  (assert (or (and cc (= c (not b))) (not cc)))

;;External Clock Constraints
7  (assert (= cclk (or ca ce)))
8  (assert (ite (and cclk (= true clk)) (= ca true)(= ca false)))
9  (assert (ite (and cclk (= false clk)) (= ce true)(= ce false)))

;;Non-Triviality Condition
10 (assert (= (or ca ce cclk cd cc) true))

;;Causal loop condition
11 (assert (and cb cc cd M1dep2 M2dep2))
(check)
```

(a) Spec1 : A false causal loop

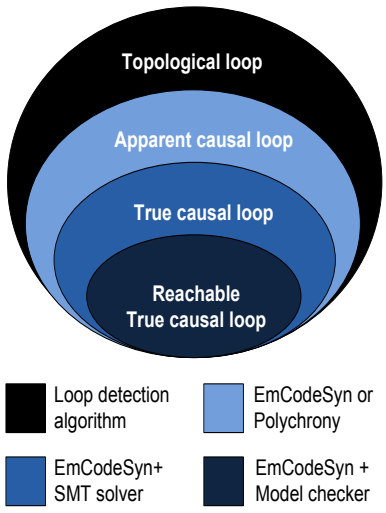
```
::S1  a := ain when trigger
1 (assert (ite (and cain ctrigger (= trigger true)) ca (not ca) ))
2 (assert (or (and ca (= a ain)) (not ca) ))

::M1  b := a default c
3 (assert (ite (or ca cc) cb (not cb)))
4 (assert (or (and cb ca (= b a)) (and cb cc (not ca) (= b c))(not cb)))
5 (assert (ite ca M1dep1 (not M1dep1) ))
6 (assert (ite (and cb cc (not ca)) M1dep2 (not M1dep2)) )

;;F1  c := b+d
7 (assert (= cc cb))
8 (assert (= cc cd))
9 (assert (or (and cc (= c (+ b d) )) (not cc) ))

;;F2  f2out  := ain >5 and ain <2   :D1
10 (assert (= cf2out cain))
11 (assert (or (and cf2out (ite (and (> ain 5) (< ain 2))
              (= f2out true) (= f2out false))) (not cf2out)))

;; S2 s2out := false when f2out    : D1
12 (assert (ite (and true cf2out (= f2out true)) cs2out (not cs2out)))
13 (assert (or (and cs2out (= s2out false)) (not cs2out) ))

;; M2 trigger := s2out default true  :D1
14 (assert (ite (or cs2out true) ctrigger (not ctrigger) ))
15 (assert (or (and ctrigger cs2out (= trigger s2out))
       (and ctrigger true (not cs2out) (= trigger true) ) (not ctrigger)))

;;External Clock Constraints C1
16 (assert (= cain cd))
17 (assert (= cain ctrigger))
;;Non-Triviality Condition
18 (assert (= (or ca cb cc cd cain ctrigger cs2out cf2out) true))
;;Causal loop condition
19 (assert (and cb cc M1dep2))
```

(a) Spec2 with constraints C1 & D1 : A true causal loop

Figure 9.4: Distinguishing between causal loops using SMT solvers

```
process Spec3 =
( ? integer ain, d;
  ! integer b; )
(| a := ain when trigger   %S1%
 | b := a default c         %M1%
 | c := b + d               %F1%
 | ain  ^= d ^=  trigger    %C1%
 | x := (x$ init 1 + 1) when ^ain default 0
 | y := (y$ init 0 + 1) when ^d   default 0
 | trigger := false when ( x == y) default true
 |)
 where integer a, c, x, y;
       boolean trigger;
end;
```

(a) Spec3: Polychronous specification for reachability test



(b) Causal loop types and tools used for their detection

Figure 9.5: Reachability test for polychronous specification and causal loop types

and EmCodeSyn are capable of identifying only up to apparent causal loops. EmCodeSyn is now automated with SMT solvers and model checkers to verify the presence of true causal loop and reachable true causal loop respectively.

9.2.2 Reachability of true causal loops

A recent work has shown how to embed polychronous specifications into synchronous ones in the form of MRICDF/SIGNAL to Quartz conversion [47]. The Averest tool was used to generate smv files from Quartz programs to determine if causal loops are constructive. A few of the Quartz representations for MRICDF actors are shown in Figure 9.6. MRICDF clock inputs are always represented as read only inputs (denoted by ?) from the environment, while MRICDF output signals are treated as read-write type, since they are written and read from modules. The imperative style code has the functionality of an actor (say `MergeInt`) expressed as epoch and data constraints. Primitive actors are instantiated, while clock or non-triviality conditions are expressed as 'assume' statements. More information on embedding polychronous processes into synchronous modules is available in [47].

The example `Spec2` with constraint $C1$ can be represented as a Quartz program as shown in Figure 9.7 (a). The standard actors $\mathtt{M}_1$ and $\mathtt{S}_1$ are instantiated from a library and the custom $\mathtt{F}_1$ actor is part of the 'loop' in Quartz code. The clock constraints and non-triviality conditions are represented as 'assume' statements, while the apparent causal loop is represented as 'assertion $A1$'. This property 'E F causal loop condition' would ask if the causal loop is a true causal loop

```
module BufferBool(
   event bool ?clk_i, ?i, ?clk_o, o,
   bool ?init)
{  bool q = init
   loop {
      if (clk_i) {
         next(q) = i;
         o = q;
      }
      assume(clk_o == clk_i);
      pause;
   }
}
```

(a) Buffer Boolean

```
module MergeInt(
   event bool ?clk_i1, event int{128} ?i1,
   event bool ?clk_i2, event int{128} ?i2,
   event bool ?clk_o,  event int{128} o,
   event bool dep1, dep2  )
{  loop {
      if (clk_i1) {
         o = i1;
         dep1 = true; dep2 = false; }
      else if (clk_i2) {
          o = i2;
          dep1 = false; dep2 = true; }
      }
      assume ((dep1 & dep2) == false);
      assume( clk_o ==  (clk_i1 | clk_i2));
      pause;
   }
}
```

(b) Merge Integer

```
module SamplerInt (
   event bool ?clk_i1, event int{128} ?i1,
   event bool  ?clk_i2, ?i2,
   event bool?clk_o, event int{128} o)
{  loop {
      if(clk_i2 & i2 & clk_i1){
          o = i1;
          }
    assume((clk_i2&i2&clk_i1) == clk_o);
    pause;
   }
}
```

(c) Sampler Integer

Figure 9.6: Quartz modules for MRICDF/SIGNAL primitives

and if it can be reached. Its equivalent representation $A2$ has been found to be more time efficient with the Cadence SMV model checker [18]. Reachability is not an issue for `Spec2` since there are no buffers in the specification. There are no initial conditions and no transitions between states for `Spec2`. In `Spec3`, where `trigger` is a computation based on a buffer output, both initial conditions and possible transitions matter. Figure 9.7 (b) shows a shortened Quartz representation of `Spec3`. The same assertion is posed as the model checking problem and it is verified that the true causal loop is not reachable with the given initial conditions. It was also verified that with identical initial values on `x` and `y`, or with favorable increment counts, the true causal loop was reachable.

The relative advantage of model checking approach as opposed to SMT approach is in reachability analysis of the causal loop state. Given a set of initial conditions, our MRICDF/Quartz transformation tool provides an infrastructure to see if the apparent causal loop is a true causal loop and also if it is reachable. The model checking time for `Spec2` amounts to a few seconds, whereas SMT technique can perform true casual loop evaluation for `Spec2` in a matter of milliseconds. Table 9.1 summarizes the time required to detect causal loops in MRICDF networks. [87] contains more information on the functionality of each MRICDF network. There are no restrictions on Integer variable size, but certain operations such as multiplications are not possible in SMT format. The number of actors in each network, the Boolean and Integer buffer counts are shown in columns 2,3, and 4 respectively. We add a simple 2 actor false causal loop (FCL) to each of these networks for computing the FCL detection time. Similarly a true causal loop (TCL) is added to the network and their detection times are reported in columns 5 and 6. Now we perform model checking on each MRICDF network with Integer variables capable of representing [-128,127] (Range1 - column 7) and [-512,511] (Range 2 - column 8). We can observe that for a given range, SMT technique has

```
import Signal.*;
module Spec2 (
   event bool ?clk_a, event int{128} a,
    ...
   event bool ?clk_trigger, event bool ?trigger,
   event bool M1dep1, M1dep2  )
{ M1: MergeInt (clk_a, a, clk_c, c, clk_b, b, M1dep1, M1dep2);
  ||
  S1: SamplerInt(clk_ain, ain, clk_trigger, trigger, clk_a, a);
  ||
  loop{
      //F1 functionality
      if(clk_b)
          c = b+d;

 assume(clk_b ==  clk_c); //F1 constraint
 assume(clk_c == clk_d); //F1 constraint
 assume(clk_ain == clk_d);// clock constraints 1
 assume(clk_d == clk_trigger);// clock constraints 1
 //assume(clk_trigger -> trigger); // uncomment for constraint2

 //Non-Triviality condition
 assume( (clk_a |clk_b |clk_c |clk_d |
         clk_ain |clk_trigger) == true);
 pause;
  }
}
satisfies {
  A1: assert E F (clk_b & clk_c & M1dep2);
  A2: assert A G ((!clk_b) | (!clk_c) | (!M1dep2));
 }
```

(a) Generated Quartz program for Spec2

```
import Signal.*;
module Spec3 (
        event bool ?clk_a, event int{128} a,
        ..
        event bool M4dep1, M4dep2, .., M1dep1, M1dep2 )
S1: SamplerInt(cain, ain, ctrigger, trigger, ca, a);
||
M1: MergeInt(clk_a, a, clk_c, c, clk_b, b, M1dep1,M1dep2);
||
B1: BufferInt(clk_$x, $x, clk_PlusX, PlusX, 1);
||
B2: BufferInt(clk_$y, $y, clk_PlusY, PlusY, 0);
..
loop{  if( clk_c ) //F1
            c = b+d;
              assume ( clk_c ==   clk_b );
              assume ( clk_c ==   clk_d );

         if( clk_x ) //F2
               {if (x ==y) xyeq = true;}
              assume ( clk_x ==   clk_y );
              assume ( clk_x ==   clk_xyeq );
           ..
       assume(clk_ain == clk_d);// clock constraints 1
       assume(clk_d == clk_trigger);// clock constraints 1
       //Non-Triviality condition
       assume( (clk_a | clk_b | …. clk_trigger) == true);
       pause;
       }
}
satisfies {
            A1: assert A G ( (!clk_b) | (!clk_c) | (!M1dep2) );
}
```

(b) Generated Quartz program for Spec3

Figure 9.7: Quartz representation of Spec2 and Spec3

MRICDF Network	**♯ of actors**	**Buffer Count**		**SMT - Time in sec**		**MC - Time in sec**	
		Boolean	**Integer**	**FCL**	**TCL**	**Range 1**	**Range 2**
Height Supervisor	7	0	0	0.109	0.094	0.188	0.844
Absolute	10	0	0	0.094	0.093	0.219	5.219
Factorial	10	2	0	0.094	0.094	5.156	98.078
Resettable Counter	10	1	0	0.93	0.109	0.109	0.266
Watchdog Timer	16	2	0	0.125	0.109	2.656	16.391
pEHBH	16	1	2	0.125	0.125	0.25	0.234
pSELF	34	6	3	0.203	0.188	1.89	7.859

Table 9.1: Time required to detect causal loops in MRICDF networks

significantly better TCL detection time. As the range is increased further, there is a longer wait time which is not desirable for an interactive programming tool. In conclusion, it is our opinion that performing SMT based causality analysis is more efficient in distinguishing between true and false causal loops. Also, only on confirmation of a true causal loop, reachability analysis before synthesis is justified.

The heuristic implemented in EmCodeSyn to fully automate our causality analysis improvements is shown in Algorithm 1. The goal of this approach is to obtain the best causality detection result with minimal analysis time. Once apparent causal loops are detected by EmCodeSyn, the MRICDF network M is converted to SMT formula S_M. The causal loop candidates are also converted as SMT assertions S_C. Now *True causal loop detection procedure* is applied using an SMT solver to see if a given candidate is a true causal loop. If a Satisfiable result is obtained, it is added into C_R for further reachability analysis. If there are true causal loops in C_R, model checking has to be performed. Each MRICDF network M with its initial conditions $init$ is converted into smv formula. The true causal loop under test is posed as a reachability problem. If the relevant property is verified to be true, then the true causal loop is declared as a reachable true causal loop. This heuristic ensures that the false causal loop identification feature of SMT technique is applied so as to minimize analysis time. Reachability analysis is performed only when a true causal loop is identified.

9.3 Software Synthesis from MRICDF specification using SMT solvers

Beyond causal loop detection, the sequential synthesis conditions for MRICDF include master trigger detection and follower set identification. A follower set is found by a repeated application of *Test For Master Trigger Signal* with a simplification process with the master trigger variables set to $true$. The master trigger detection test using a prime implicate generator is the most time consuming step in EmCodeSyn design flow. In this section, we discuss how SMT solvers are used

Algorithm 1: *To find reachable true causal loops*

```
/* Find C, the set of apparent causal loops in M                    */
C = Find_ApparentCL(M)
if C ≠ φ then
    /* Convert MRICDF M to SMT formula S_M                          */
    S_M = Convert_MRICDFtoSMT(M)
    S_C = Convert_MRICDFtoSMT(C)
    /* Test satisfiability of the SMT formula with causal loop
    assertion                                                        */
    for S_C^i ∈ S_C do
        if S_M, S_C^i ⊨ true then
            /* Add true causal loop C^i to the set C_R for reachability
            analysis                                                 */
            C_R ← C_R • C^i
        end
    end
end
if C_R ≠ φ then
    /* Convert MRICDF M to SMV V_M                                  */
    V_M, t_init = Convert_MRICDFtoSMV(M, init)
    /* Generate causal loop property SPEC (A G (not (C_R )))        */
    P_R = Generate_CausalLoopProperties(C_R)
    /* Model checking for causal loop C_M^i on MRICDF model M with
    initial state                                                    */
    for P_R^i ∈ P_R do
        if V_M, t_init ⊨ P_R^i then
            /* Add C_M^i as a reachable true causal loop in M       */
            C_L ← C_L • C_M^i
        end
    end
end
```

to perform the master trigger test to speed up software synthesis.

9.3.1 Master trigger test

The test for identifying master trigger signal is based on Theorem. 6.3.1, which uses the property of a master trigger signal, i.e., if a master trigger signal is absent, no other signal can be present. The SMT method for performing this test is to encode the test as a satisfiability problem. There is a constraint added (*at least one clock variable has to be true*) to avoid the trivial solution (*all Boolean variables are* $false$). Since a root clock signal has to be present for every instant, setting root clock Boolean variable to $false$ will turn every Boolean variable to $false$, indicating the absence of an instant without root clock event. This trivial solution has already been avoided, thus a contradiction (UNSAT result) is obtained from an SMT solver.

For an MRICDF network consisting of signals $x_1, x_2, ..., x_n$, the *MRICDF-SMT translation* is performed to convert epoch equations to SMT assertions as given in Figure 9.3. A *non-triviality condition* (here `assert (= true (or cx1 cx2 ... cxn))`) is added to remove the solution where all variables are *false* (i.e. $cx_1 \cup cx_2 \cup .. \cup cx_n = false$). Finally, a *master trigger condition* is added, where the master trigger candidate signal is set to $false$. The solutions to the SMT formula are the various combinations of values to the clock variables, representing all possible outcomes in various instants of the MRICDF network. To summarize, the SMT method of master trigger testing has two possible results:

1. If the constructed SMT formula is **satisfiable**, there is no conflict between non-triviality condition and master trigger condition. This implies there is a valid instant with a clock signal other than the master trigger candidate. So the candidate is **not the master trigger**.
2. If the constructed SMT formula is **unsatisfiable**, there is a conflict between non-triviality condition and master trigger condition. Master trigger signal being false, forced every other signal to be false, and hence the candidate **is the master trigger**.

Implementability of endochronous specification

An endochronous polychronous specification is one which has a hierarchic clock-tree with a root clock. Listing 7.1 in Chapter 7 discussed one such polychronous sample program in SIGNAL (`Absolute`), where the absolute value of an input signal $inval$ was computed. Figure 9.8 is the screen shot of the MRICDF network `Absolute`, in the EmCodeSyn environment.

Shown in Listing 9.1 is the SMT formula for testing master trigger candidates. Other than signal value and clock definitions, the SMT representation is split into the three parts: MRICDF-SMT actor translation, non-triviality condition and master trigger condition. An unsatisfiable result is obtained for $cinval$ (`F1i1`) and many other clock signals. They constitute the first element of the follower set $\{inval, absol, neg, mult\}$. Once the clock signals in the first element are set to true

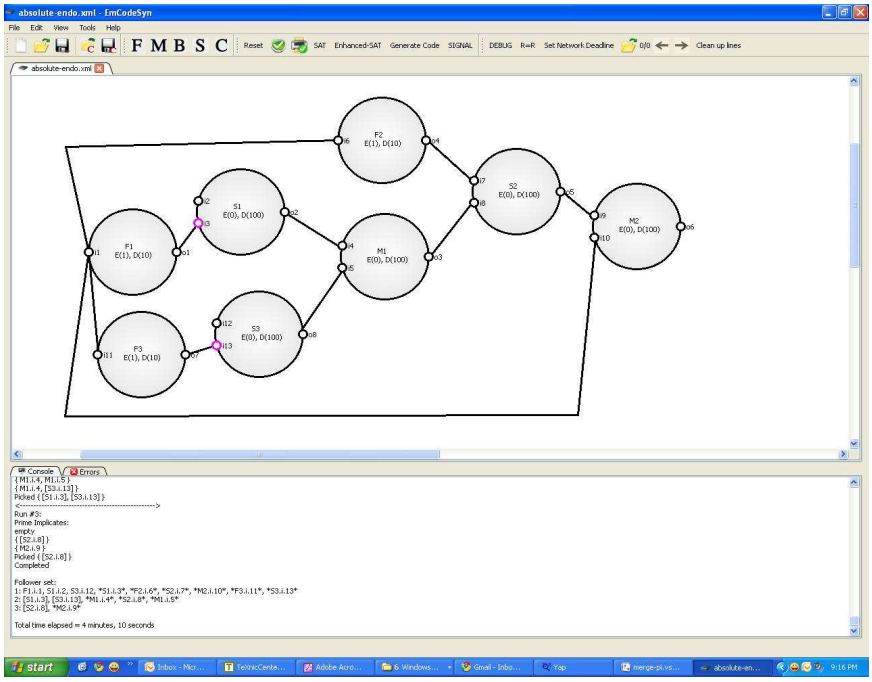


Figure 9.8: MRICDF network `Absolute` designed in EmCodeSyn environment

and master trigger test is repeated, subsequent elements are obtained, proving the specification to be endochronous.

Listing 9.1: SMT representation for master trigger identification.

```
(reset)
(set-evidence! true)
(set-verbosity! 3)
(define-type l_int (subtype (n::int) (and (> n -101) (< n 101))))

;;Open Inputs
(define F1i1 :: l_int)
(define cF1i1 :: bool)
..
(define cF3i11 :: bool)

;;Open Outputs
(define M2o6 :: l_int)
(define cM2o6 :: bool)

;;Data Dependency Variables
(define M2dep1 :: bool)
(define M2dep2 :: bool)
(define M1dep1 :: bool)
(define M1dep2 :: bool)

;;Internal Variables
(define S2i7 :: l_int)
  ....
(define cS3o8 :: bool)

```

```
27 ;; Function: F1 -> inval < 0
28 (assert (= cF1o1 cF1i1))
29 (assert (or (and cF1o1 )(not cF1o1)))
30 (assert (= cF1i1 cF2i6))
31 (assert (= cF1i1 cM2i10))
32 (assert (= cF1i1 cF3i11))
33
34 ;; S1o2 := S1i2 when S1i3  ->  false when (inval >0)
35 (assert (ite (and cS1i2 cS1i3 (= S1i3 true)) cS1o2 (not cS1o2)))
36 (assert (or (and cS1o2(= S1o2 S1i2))(not cS1o2)))
37 (assert (= cS1i3 cF1o1))
38
39 ;; Function: F2  -> inval * -1 //multiplication not performed
40 (assert (= cF2o4 cF2i6))
41 (assert (or (and cF2o4 )(not cF2o4)))
42
43 ;; S2o5 := S2i7 when S2i8 ->  (mult when neg)
44 (assert (ite (and cS2i7 cS2i8 (= S2i8 true)) cS2o5 (not cS2o5)))
45 (assert (or (and cS2o5(= S2o5 S2i7))(not cS2o5)))
46 (assert (= cS2i7 cF2o4))
47 (assert (= cS2i8 cM1o3))
48
49 ;; M2o6 := M2i9 default M2i10 -> absol:=(mult when neg) default (inval when (not neg))
50 (assert (ite (or cM2i9 cM2i10) cM2o6 (not cM2o6)))
51 (assert (or (and cM2o6 cM2i9 (= M2o6 M2i9))(and cM2o6 cM2i10(not cM2i9)(= M2o6 M2i10))
52                                                  (and (not cM2o6)(not cM2i9)(not cM2i10))))
53 (assert (ite cM2i9 M2dep1 (not M2dep1) ))
54 (assert (ite (and cM2o6 cM2i10 (not cM2i9)) M2dep2 (not M2dep2)))
55 (assert (= cM2i9 cS2o5))
56
57 ;; Function: F3 -> inval >= 0
58 (assert (= cF3o7 cF3i11))
59 (assert (or (and cF3o7 )(not cF3o7)))
60
61 ;; M1o3 := M1i4 default M1i5 -> neg:= true when(inval <0) default false when(inval >0)
62 (assert (ite (or cM1i4 cM1i5) cM1o3 (not cM1o3)))
63 (assert (or (and cM1o3 cM1i4 (= M1o3 M1i4))(and cM1o3 cM1i5(not cM1i4)(= M1o3 M1i5))
64                                              (and (not cM1o3)(not cM1i4)(not cM1i5))))
65 (assert (ite cM1i4 M1dep1 (not M1dep1) ))
66 (assert (ite (and cM1o3 cM1i5 (not cM1i4)) M1dep2 (not M1dep2)))
67 (assert (= cM1i4 cS1o2))
68 (assert (= cM1i5 cS3o8))
69
70 ;; S3o8 := S3i12 when S3i13 -> true when (inval < 0)
71 (assert (ite (and cS3i12 cS3i13 (= S3i13 true)) cS3o8 (not cS3o8)))
72 (assert (or (and cS3o8(= S3o8 S3i12))(not cS3o8)))
73 (assert (= cS3i13 cF3o7))
74
75 ;;Triviality condition
76 (assert (or cF1i1 cF2i6 cM2i10 cF3i11))
77
78 ;;Causal Loop condition
79
80 ;;Master trigger condition
81 (assert (= false cF1i1))
82 (check)
```

Implementability of non-endochronous specification

A non-endochronous SIGNAL specification is one which does not have a hierarchic clock tree with root clock. A simple example for a non-endochronous specification is the `Merge` actor (say $z = x\ default\ y$). It is non-endochronous, since there is insufficient information on the input clock signals and hence their order of reading is non-deterministic. Using the SMT synthesis technique, it can be seen that the formula for non-endochronous actors `Merge` and `Sampler` is satisfiable with any clock signal put as a master trigger condition. Whereas the SMT formula for endochronous actors `Buffer` and `Function` become unsatisfiable when any of their input signals is tested as the master trigger candidate.

Now we show how the exogenous information (external reference clock is master trigger) is put into non-endochronous `Merge` actor and how the SMT formula is transformed to give an UNSAT result for the external clock signal. Both Polychrony and EmCodeSyn are capable of identifying non-endochronous specification and ask for external information or epoch constraints. In Polychrony compiler, the specification is made endochronous by adding exogenous information. In the case of `Merge` actor, two additional signals $c1, c2$ are added having the same clock cc. Each of these signals have a true part ($c1t, c2t$) and a false part ($c1f, c2f$) respectively. In Boolean logic, $c1 = c1t \vee c1f$, and $c2 = c2t \vee c2f$. Also these parts are mutually exclusive, ie a signal cannot be *true* and *false* at the same time. Hence, $c1t \wedge c1f = false$ and $c2t \wedge c2f = false$. Now the signals $c1t$ and $c2t$ are assigned as clock signals for x and y respectively as shown in Figure 9.9. The signal clocks remain different ($c1t \neq c2t$), but their parent clock is the same cc. The false part of clock signals such as $c1f, c2f$ is avoided, as no data or clock operations can happen. The SMT representation of Merge actor with exogenous information is shown on the left hand side. Only the candidates $c1 = false$ (in listing), $c2 = false$ give an unsatisfiable result. Thus, the master trigger signal set is $\{c1, c2\}$. The follower set is found by setting these signals to $true$, simplifying the set of formula and repeating master trigger identification step. $cx = c1t$ and $cy = c2t$ is the signals with the highest epoch, second time around. The follower set is as follows: FS= $\{\{c1, c2\}, \{(c1t + c2t), (cx + cy), cz\}\}$.

9.4 Comparison of two implementations for Boolean theory approach to synthesis

We compare the master trigger detection time for different MRICDF networks in Table 9.2. Earlier PI based implementation of our Boolean theory approach had high synthesis time as shown in column 3 (Original) of Table 9.2. For non-endochronous polychronous specification, the time to report the absence of master trigger is shown in the table. Our actor elimination technique (AET) [86] reduced the synthesis time by giving optimized Boolean equations to the time consuming PI generator. The alternative SMT methodology at the Boolean level outperforms both these enhancements by several orders of magnitude as shown in the column 5 (SMT). Polychrony tool takes at

```
;;merge z:= x default y
(assert (ite (or cx cy) cz (not cz) ) )
(assert (or (and cz cx (= z x))
      (and cz cy (not cx) (= z y)) (not cz)))

;;define exogenous information
(assert (= c1 c2) )

;;c1= c1t OR c1f, c1t AND c1f = false
(assert (= c1 (or c1t c1f) ) )
(assert (= false (and c1t c1f) ) )

;;c2 = c2t OR c2f, c2t AND c2f = false
(assert (= c2 (or c2t c2f) ) )
(assert (= false (and c2t c2f) ) )

;; cx := true when c1t=true
;; cy := true when c2t=true
(assert (ite (= c1t true) (= cx true) (not cx ) ) )
(assert (ite (= c2t true) (= cy true) (not cy ) ) )

;; Non-Triviality condition
(assert (= (or  c1 c2 cx cy cz ) true) )

;; master trigger condition
(assert+ (= (or cy cx) false ) )
```

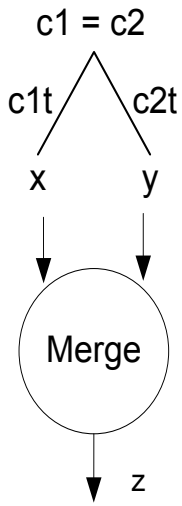


Figure 9.9: The synthesizable `Merge` with exogenous information

MRICDF Network	**♯ of actors**	**Time in seconds**		
		Original	**AET**	**SMT**
Height Supervisor	5	0.35	0.37	0.094
Absolute	8	0.91	0.91	0.078
Factorial	8	0.33	0.33	0.09
Resettable Counter	8	0.927	0.562	0.099
Watchdog Timer	14	16.3	5.3	0.11
Producer-Consumer	15	21.4	16.3	0.12
Flight Warning System	17	8.1	1.03	0.1
GCD	19	1.35	0.89	0.13
pEHBH	14	0.79	0.75	0.125

Table 9.2: Time required to find Master Trigger signal

most a few seconds for compilation for all the above mentioned examples. It is difficult to separate Polychrony clock hierarchy determination time and we do not claim EmCodeSyn to be faster. For EmCodeSyn with prime implicate approach, it is observed that the synthesis time increases for certain large examples, thus harming the practical use of the tool. SMT based master trigger detection is observed to be taking only milliseconds irrespective of program size, thus bringing down synthesis time for EmCodeSyn. More details on EmCodeSyn code generation are given in [95]. In conclusion, embracing the SMT route for implementing Boolean theory based software synthesis is beneficial in improving the causality analysis of polychronous programs as well as the software synthesis time.

Chapter 10

Conclusions

Model driven software synthesis is gaining traction in the industry due to its relative benefits such as higher productivity, lower time to market, compatibility with verification tools, etc. Automated code generation from a specification captured at a higher level abstraction enables the user to optimize each implementation. Our work explored opportunities in the design, synthesis and composition of embedded software specified in polychronous formalism. The background information regarding polychronous formalism and software synthesis from polychronous specifications were discussed in Chapter 2. Chapter 3 discussed the existing compilers and synthesis tools from synchronous specifications and their multi-rate versions. The sequential and parallel programming models employed were also discussed.

10.1 Programming models for polychronous software synthesis

Our research started with a study of the existing synthesis tools for polychronous software. Polychrony was used extensively to generate implementation level code and to simulate their behavior. Embedded software targeting Globally Asynchronous and Locally Synchronous (GALS) architecture was of particular interest. Endochronous SIGNAL programs which have a unique order of execution were suitable for synchronous local modules. But the composition of these modules was difficult, due to the strict clock relations that had to be maintained between communicating signals. Weakly endochronous SIGNAL programs which allow local non-determinism that does not impair global determinism were found to be much more conducive to composition. We proposed assertions for isochronous composition, which requires communicating synchronous components to agree on the values of shared signals at each time instant. A communication structure was defined which consisted of minimal information that has to be shared between communicating polychronous modules. Composition of endochronous and weakly endochronous programs in SIGNAL were analyzed and our proposed assertions for testing proper isochronous composition were shown to be true. These research results were presented in Chapter 4.

Understanding software synthesis criteria and composition of modules involved studying programming models followed by these synthesis tools. Of particular interest was the multi-threading code generation approaches from a language such as SIGNAL compiler, which can capture concurrent specifications. Lessons learned and suggested improvements are summarized in Chapter 5. Weak endochrony gave reasons for pursuing multi-threading even further, due to the freedom in non-deterministic local behavior allowed in a module. We proposed the use of synchronous flow dependence graphs to infer a threading structure from SIGNAL programs. The existing micro-threading strategy which spawns new threads for each signal had considerable overhead when scaled to larger designs. An alternative suggested was to operate each module of a specification as a thread. SIGNAL processes were simulated as coarse grained threads with mutual exclusion primitives for shared signals. In hindsight, we moved away from our early works in chapters 4 and 5 on existing polychronous compilers, towards creating our own formalism, sequential synthesis criteria and synthesis strategy. Now, with these contributions implemented in our own software synthesis tool, one can revisit contributions in composition and multi-threading for integration into the new framework.

10.2 Our contributions towards sequential software synthesis

We proposed an alternative to SIGNAL formalism and Polychrony's clock calculus in the form of MRICDF visual formalism and Epoch Analysis in Chapter 6. With MRICDF, a visual formal model based on actor interactions was realized for capturing polychronous specification. The notion of epoch was introduced, which is a history of occurrences of events on a signal. Standard actors were defined, which performed operations from set theory such as intersection and union on signal epochs. A novel synthesis strategy (Epoch Analysis) was proposed which enabled the user to perform sequential implementability analysis at the Boolean level, instead of directed graphs used by Polychrony. In this approach, epoch relations between signals were converted to Boolean equations where each Boolean variable represented the presence or absence of an event at an arbitrary instant. The system of Boolean equations could have many solutions which were representative of the multiple instants of the MRICDF network. The goal of epoch analysis was to find the signal whose epoch is the superset of any other signal epoch in the MRICDF network. This particular signal is called the master trigger signal, whose will have a synchronous event with every other signal event of the MRICDF network. Thus a total order on instants of the MRICDF network was established, since the master trigger signal is a reference clock tick for each instant. Now a unique order of execution has to be identified, called the follower set. A repeated master trigger identification process with the knowledge of data flow between signals was found to be sufficient for finding a unique order of execution.

A prime implicate (PI) clause of a Boolean system of clauses will have to be true for every solution. Another contribution was an analogy shown between master trigger signal and a single positive PI literal among PIs. Since every interpretation of the Boolean system is representative of an instant of MRICDF network, the master trigger signal has to be always present. PIs are also clauses which

have to be always true. For MRICDF Boolean system of equations, this implies the associated signal for the single positive literal PI is a master trigger. Similar properties were identified for generating the follower set. This contribution paved the way for developing a software synthesis tool which eliminates the need for construction of a clock tree of signals to check SIGNAL implementability conditions: hierarchic clock tree with a root clock signal. Chapter 7 explains the prime implicate based technique of implementing Epoch Analysis for MRICDF network. Our software synthesis tool EmCodeSyn provides a GUI for the user to specify polychronous actor networks with epoch constraints and customized functions. Code generation can take place in case of favorable implementability results. The generated C code can be used to simulate the MRICDF specification with input-output files giving epoch and data information of signals.

10.3 Optimizations for speeding up polychronous software synthesis

During the development of EmCodeSyn, one of the major concerns was the long prime implicate computation time. An external prime implicate generator was interfaced with EmCodeSyn to provide prime implicates of the system of Boolean equations. Newer versions of prime implicate generator were interfaced to reduce PI computation time. Further reduction in synthesis time was possible only by giving a smaller set of Boolean equations to the PI generator. We proposed the Actor Elimination Technique (AET) for the removal of actors which do not play a role in the formation of master trigger epoch. This contribution is covered in Chapter 8. Actors that do not alter signal epochs were eliminated along with those having lower signal epochs. AET rules were drawn up for keeping the master trigger epoch of the original and simplified MRICDF networks the same. Theorems and proofs were provided to show AET rules and their application do not alter the follower set of an MRICDF network.

Lowering synthesis time using AET is dependent on the nature of the MRICDF network under consideration. Presence of more epoch modifying actors do not help elimination and hence other alternatives were explored. Recent advances in the use of verification tools for synthesis purposes led us to trying out SAT/SMT solvers as a replacement for PI generator. We translated the Boolean equations to assertions compatible with an SMT solver. Setting the master trigger candidate variable to false challenged SMT solver to find an instant where master trigger candidate does not have an event. Since such an event is not possible for a master trigger signal, the SAT/UNSAT results were interpreted as master trigger test results. We have shown our translation procedure for MRICDF actors with theorems and proofs to supplement our proposal in Chapter 9. SMT solvers provided us with flexibility to make intelligent choices for master trigger candidates. This method was found to be much faster than using a PI generator for master trigger identification.

10.4 Broadening the set of implementable polychronous specifications

The development of MRICDF formalism and EmCodeSyn allowed us to have a closer look at the boundaries drawn by existing synthesis tools for judging what are acceptable programs. We believe Polychrony has followed a conservative approach to not accept specifications which might have multiple behaviors. The lack of data information encoded into intermediate representation format has resulted in rejection of programs with data constraints which force a unique behavior. Determining a legitimate boundary for the set of implementable polychronous specifications was a problem addressed in Chapter 9. Also causal loop analysis, a sequential synthesis condition for any polychronous synthesis tool was found to be insufficient for distinguishing between causal loop types. Causal loop analysis techniques for other synchronous languages such as Esterel or Lustre follow clock constraint analysis or model checking based approaches to accept or reject programs. Existing compilers looked for structural buffer free cycles in specification that can be constructive in the event of favorable clock constraints. Taking a leaf out of our 'satisfiability-for-synthesis' contribution, we posed each causal loop condition as a satisfiability problem. With encoding of data enabled by our MRICDF-SMT translation, checking for data dependent true causal loops were made possible. This was shown to be capable of distinguishing false causal loops, which were the cause for wrongly rejected programs. Our causality analysis approaches were found to be time-efficient and capable of detecting each loop type, except for its reachability analysis.

The need for reachability analysis spawned a related work, not included in this dissertation, where embedding of polychronous modules into synchronous ones was realized. MRICDF modules were implemented as Quartz programs which can be simulated and used for verification purposes. Model checking capabilities of Averest environment which accepts Quartz programs were utilized to perform reachability analysis for programs with true causal loops. With SMT based false causal loop detection included in EmCodeSyn, minimal use of the time consuming model checking step was required. These contributions in causal loop analysis, while applied on EmCodeSyn, are compatible with Polychrony.

10.5 Future work

There were several works in this dissertation that had co-authorship of researchers in the area of embedded software synthesis. Embedding polychronous modules into Quartz programs is a related work triggered due to our investigations into causality loop analysis. Development of EmCodeSyn will continue to throw challenges in the areas were extensions are being planned. An ongoing work relating to EmCodeSyn is composition of MRICDF networks where both clock and data relations are considered. We have proposed assertions which can determine when two polychronous specifications are compatible for composition with respect to their clock relations. Currently a composite MRICDF actor is unrolled for Epoch Analysis, thus exposing its IP information to users. Protect-

ing the IP and performing a non-invasive compatibility analysis would require understanding of the minimal information that needs to be provided with each IP. One of the researchers is investigating these possibilities and will be continuing to add such features into EmCodeSyn. Another researcher is trying to encode an Unmanned Areal Vehicle (UAV) as a polychronous specification. The goal is to synthesize embedded software for its target platform and to identify optimizations that are made possible due to multi-rate specifications.

The causality analysis of MRICDF networks used an SMT solver for validating presence of a true causal loop. Reachability analysis was required to accept more networks. A model checker was used for this purpose. An alternative to this approach is the use of Bounded Model Checking using SMT solvers. Since an SMT solver cannot express memory elements, each consecutive instants have to be iterations of the SMT solver. An inductive proof or a counter example can accept or reject the presence of a causal loop. This would be a possible extension of the work done using SMT solvers.

Ergonomic and aesthetic value of a software tool is crucial in promoting its use among new users. We have been working on enhancing the appearance of MRICDF networks at each step of the design flow. Several synthesis strategies have been applied on MRICDF networks including synchronous data flow, prime implicate approach, SMT approach etc. The Network Information File of each MRICDF example has been revised to provide the additional information required by each approach. We have experimented with generation of SIGNAL code from MRICDF networks. Another path explored was the scheduling of actors, given a fixed deadline for an instant. The worst case execution time analysis for polychronous specifications would be an important extension. It would be more intuitive for MRICDF actor networks as opposed to the SIGNAL textual form. These are some of the research directions that are in progress. In this dissertation, we have discussed existing multi-threading approaches for polychronous software, but we did not explore multi-threaded code generation from MRICDF formalism. We believe our contributions have given a platform to extend this work into new domains and to spur new ideas in the area of software synthesis.

10.6 Publications on the work reported in this dissertation

The following are the peer-reviewed publications and research reports on the work done in this dissertation. All the work in terms of the research contributions, implementation and experimentation for these publications was done by the author under the guidance of Dr. Sandeep K. Shukla. The technical reports on currently unpublished works can be found at web page of FERMAT Lab at Virginia Tech (http://www.fermat.ece.vt.edu/).

1. Bijoy A. Jose, Hiren D. Patel, Sandeep K. Shukla, and Jean-Pierre Talpin, Generating Multi-Threaded code from Polychronous Specifications, *Electronic Notes in Theoretical Computer Science, (SLAP)*, vol. 238, pp. 57-69, 2008.

2. Bijoy A. Jose, Sandeep K. Shukla, Hiren D. Patel, and Jean-Pierre Talpin, On the Deterministic Multi-threaded Software Synthesis from Polychronous Specifications, *In the Proc. of the 6th ACM/IEEE International Conference on Formal Models and Methods in Co-Design (MEMOCODE)*,pp. 129-138, Anaheim, CA, June 2008.

3. Bijoy A. Jose, Bin Xue, and Sandeep K. Shukla, An analysis of the composition of synchronous systems, *Electronic Notes in Theoretical Computer Science, (FMGALS)*, vol. 245, pp. 69-84, 2009.

4. Bijoy A. Jose, Lemaire Stewart, Jason Pribble, and Sandeep K. Shukla, EmCodeSyn: A Visual Environment for Multi-Rate Data Flow Specifications and Code Synthesis for Embedded Applications, *In the Proc. of the 12th Intl. IEEE Forum on specification and Design Languages (FDL)*, pp. 1-6, Sophia Antipolis, France, Sept. 2009.

5. Bijoy A. Jose, Bin Xue, Sandeep K. Shukla, and Jean-Pierre Talpin, Programming models for Multi-Core Embedded Software, *Book Chapter in "Multi-Core Embedded Systems"*, ISBN 978-1-4398-1161-0, CRC Press, Taylor & Francis, pp. 269-368, 2010.

6. Bijoy A. Jose and Sandeep K. Shukla, An Alternative Polychronous Model and Synthesis Methodology for Model-Driven Embedded Software, *In the Proc. of the 15th IEEE Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 13-18, Taiwan, Jan. 2010.

7. Bijoy A. Jose and Sandeep K. Shukla, MRICDF : A polychronous Model for Embedded Software Synthesis. *Book Chapter in: "Synthesis of embedded software: frameworks and methodologies for correctness by construction software design"*, ISBN 978-1-4419-6399-4, pp. 173-199, Springer, 2010.

8. Bijoy A. Jose, Jason Pribble and Sandeep K. Shukla, Faster software synthesis using Actor Elimination Techniques for Polychronous formalism, *in Proc. of 10th IEEE Intl. Conf. on Application of Concurrency to System Design (ACSD)*, pp. 147-156, Portugal, June 2010.

9. Bijoy A. Jose, Abdoulaye Gamatie, Julien Ouy and Sandeep K. Shukla, SMT Based False Causal loop Detection during Code Synthesis from Polychronous Specifications, *ACM/IEE 9th Intl. Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pp. 109-118, Cambridge, UK, July, 2011.

10. Bijoy A. Jose and Sandeep K. Shukla, New Techniques for Sequential Software Synthesis from a Polychronous Data Flow Formalism, *FERMAT Technical Report 2011-07*, 2011. (Paper under review)

11. Bijoy A. Jose, Abdoulaye Gamatie, Matthew Kracht and Sandeep K. Shukla, Improved False Causal Loop Detection in Polychronous Specification of Embedded Software, *FERMAT Technical Report 2011-08*, 2011. (Paper under review)

Bibliography

[1] ARM Cortex-A9 MPCore - Multicore processor. http://www.arm.com/products/CPUs/ARMCortex-A9_MPCore.html.

[2] CellBE: Cell Broadband Engine Architecture (CBEA). http://www.research.ibm.com/cell/.

[3] Columbia Esterel Compiler. http://www.cs.columbia.edu/ sedwards/cec/.

[4] EmCodeSyn: A visual software synthesis tool for polychronous specifications. http://www.fermat.ece.vt.edu/?page_id=165.

[5] IEEE POSIX standardization authority. http://standards.ieee.org/regauth/posix/.

[6] Intel Thread Building Blocks. http://www.threadingbuildingblocks.org/.

[7] Mathworks Real Time Workshop. http://www.mathworks.com/products/rtw/.

[8] Mathworks SIMULINK. http://www.mathworks.com/products/simulink/.

[9] Microsoft Windows Threads. http://msdn.microsoft.com/.

[10] National Instruments LabVIEW. http://www.ni.com/labview/.

[11] National Instruments MATRIXx. http://www.ni.com/matrixx/.

[12] NVIDIA Compute Unified Device Architecture. www.nvidia.com/cuda.

[13] Qt Cross-Platform Application Framework. www.trolltech.com/products/qt/.

[14] Renesas SH2A-DUAL SuperH Multi-Core Microcontrollers. http://www.renesas.com/media/company_info/news_and_events/press_releases/2007/0416/20070416e.pdf.

[15] RT-Builder, GEENSYS . http://www.geensys.com/.

[16] Stanford University graphics Lab, BrookGPU. http://graphics.stanford.edu/projects/brookgpu/index.html.

[17] Sun Niagara Processor. http://www.sun.com/processors/niagara/.

[18] The Cadence SMV model checker. http://www.kenmcmil.com/smv.html.

[19] The OpenMP API specification for parallel programming. http://openmp.org/wp/.

[20] The Ptolemy project. www.ptolemy.eecs.berkeley.edu/.

[21] B. Dutertre and L. de Moura. The Yices SMT solver.

[22] STARMAC Project Group. The Stanford Testbed of Autonomous Rotorcraft for Multi-Agent Control Overview. http://hybrid.stanford.edu/starmac/overview.

[23] S. Ahuja. High Level Power Estimation and Reduction Techniques for Power Aware Hardware Design. 2010.

[24] S. Ahuja, D. A. Mathaikutty, A. Lakshminarayana, and S. Shukla. Accurate power estimation of hardware co-processors using system level simulation. *22nd IEEE International SOC Conference*, pages 399–402, 2009.

[25] S. Ahuja, D. A. Mathaikutty, A. Lakshminarayana, and S. K. Shukla. SCoPE: Statistical Regression Based Power Models for Co-Processors Power Estimation. *Journal of Low Power Electronics*, 5(4):407–415, 2009.

[26] S. Ahuja, D. A. Mathaikutty, and S. Shukla. Applying verification collaterals for accurate power estimation. *9th International workshop on Microprocessor test and Verification (MTV)*, pages 61–66, 2008.

[27] S. Ahuja, D. A. Mathaikutty, S. Shukla, and A. Dingankar. Assertion-Based Modal Power Estimation. *8th International workshop on Microprocessor test and Verification (MTV)*, pages 3–7, 2007.

[28] S. Ahuja, D. A. Mathaikutty, G. Singh, J. Stetzer, S. Shukla, and A. Dingankar. Power estimation methodology for a high-level synthesis framework. *10th International Symposium on Quality Electronics Design (ISQED)*, pages 541–546, 2009.

[29] S. Ahuja and S. K. Shukla. MCBCG: Model Checking Based Sequential Clock-Gating. *IEEE International workshop on High Level Design Validation and Test*, pages 20–25, November 2009.

[30] S. Ahuja, W. Zhang, A. Lakshminarayana, and S. K. Shukla. A Methodology for Power Aware High-Level Synthesis of Co-Processors from Software Algorithms. *proceedings of International VLSI design Conference, India, 2010*, pages 282–287, January, 2010.

[31] S. Ahuja, W. Zhang, and S. K. Shukla. System level simulation guided approach to improve the efficacy of clock-gating. *15th IEEE International workshop on High Level Design Validation and Test*, pages 9–16, June 2010.

[32] T. P. Amagbegnon, L. Besnard, and P. L. Guernic. Implementation of the data-flow synchronous language signal. In *ACM Symp. on Prog. Languages Design and Implementation (PLDI'95)*, volume 1, pages 163–173, 1995.

[33] P. Aubry, P. L. Guernic, and S. Machard. Synchronous distribution of SIGNAL programs. In *Proc. of HICSS-29*, volume 1, pages 656–665, January 1996.

[34] P. Aubry, P. L. Guernic, and S. Machard. Synchronous distribution of signal programs. In *29th Hawaii Intl. Conf. on System Sciences*, volume 1, pages 656–665, 1996.

[35] T. Austin, D. Blaauw, T. Mudge, and K. Flautner. Making typical silicon matter with razor. *Computer*, 37(3):57–65, 2004.

[36] C. Barrett, M. Deters, A. Oliveras, and A. Stump. Satisfiability Modulo Theories Competition (SMT-COMP) . http://www.smtexec.org/exec/?jobs=684, 2010.

[37] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: a programming model for the cell BE architecture. *In Proceedings of the ACM/IEEE Conference on Supercomputing*, 2006.

[38] A. Benveniste, B. Caillaud, and P. L. Guernic. From synchrony to asynchrony. In *Proceedings of Concurrency Theory*, volume 1664. Springer, 1999.

[39] A. Benveniste, B. Caillaud, and P. L. Guernic. Compositionality in Dataflow Synchronous Languages: Specification and Distributed Code Generation. *Information and Computation*, 163(1):125–171, 2000.

[40] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages twelve years later. *Proc. of the IEEE :Sp. Is. on Modeling and Design of Embedded Systems*, 91(1):64–83, 2003.

[41] G. Berry. The constructive semantics of pure esterel, 1996.

[42] G. Berry and G. Gonthier. The esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87 – 152, 1992.

[43] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Sci. Comput. Program*, 19(2):87–152, 1992.

[44] G. Berry and E. Sentovich. *Multiclock Esterel: Correct Hardware Design and Verification Methods. In Lec. Notes in Comp. Sc.*, volume 2144/2001. Springer-Verlag Publishers, 2001.

[45] M. Blum and S. Kanna. Designing programs that check their work. In *Proc. of the 21st ACM symposium on Theory of computing*, pages 86–97, New York, NY, USA, 1989. ACM.

[46] M. Bozzano, R. Bruttomesso, A. Cimatti, T. A. Junttila, S. Ranise, P. V. Rossum, and R. Sebastiani. Efficient satisfiability modulo theories via delayed theory combination. In *Computer Aided Verification*, pages 335–349, 2005.

[47] J. Brandt, M. Gemunde, K. Schneider, B. A. Jose, and S. K. Shukla. Causality Analysis of Polychronous Programs. *FERMAT Technical Report 2011-02*.

[48] J. Brandt and K. Schneider. Separate translation of synchronous programs to guarded actions. Internal Report 382/11, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, March 2011.

[49] J. Brandt, K. Schneider, S. Ahuja, and S. K. Shukla. The model checking view to clock gating and operand isolation. pages 181–190, June 2010.

[50] C. Brunette, J.-P. Talpin, L. Besnard, and T. Gautier. Modeling multi-clocked data-flow programs using the generic modeling environment. In *Synchronous Languages, Applications, and Programming(SLAP'06)*, pages 3284–3289, March 2006.

[51] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From simulink to SCADE/lustre to TTA: a layered approach for distributed embedded applications. *SIGPLAN Not.*, 38(7):153–162, July 2003.

[52] P. Caspi, A. Girault, and D. Pilaud. Automatic distribution of reactive systems for asynchronous networks of processors. *IEEE Transactions on Software Engineering*, 25(3):416–427, May 1999.

[53] P. Caspi and M. Pouzet. A Functional Extension to Lustre. In M. A. Orgun and E. A. Ashcroft, editors, *International Symposium on Languages for Intentional Programming*, Sydney, Australia, May 1995. World Scientific.

[54] A. Cimatti. Beyond boolean sat: Satisfiability modulo theories. In *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*, pages 68 –73, may 2008.

[55] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[56] J. de Kleer. An improved incremental algorithm for computing prime implicants. In *Proc. AAAI-92*, pages 780–785, San Jose, CA, 1992.

[57] L. de Moura and N. Bjorner. Satisfiability modulo theories: An appetizer. *Lecture Notes in Computer Science*, 5902:23–36, 2009.

[58] E. W. Dijkstra. Cooperating sequential processes. *Communications of the ACM*, 26(1):100–106, Jan. 1983.

[59] S. A. Edwards and O. Tardieu. SHIM: A Deterministic Model for Heterogeneous Embedded System. *IEEE Trans. on Very Large Scale Integration Systems*, 14:854–867, 2006.

[60] S. A. Edwards and J. Zeng. Code Generation in the Columbia Esterel Compiler. *EURASIP J. on Embedded Systems*, 2007:1–31, 2007.

[61] ESPRESSO Project, IRISA. The Polychrony Toolset. http://www.irisa.fr/espresso/Polychrony.

[62] ESTEREL Technologies. SCADE suite. www.esterel-technologies.com/products/scade-suite/.

[63] ESTEREL Technologies report. SCADE suite. http://www.sigada.org/conf/sigada2003/SIGAda2003-CDROM/SIGAda2003-Proceedings/Vendor-Esterel-SCADE-Suite-for-Ada.pdf.

[64] A. Gamatié. *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification*. Springer-Verlag New York, 2009.

[65] A. Gamatié and L. Gonnord. Static Analysis of Synchronous Programs in SIGNAL for Efficient Design of Multi-Clocked Embedded Systems. In *Proc. of SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'11)*, pages 71–80, April 2011.

[66] T. Gautier, P. L. Guernic, and L. Besnard. SIGNAL: A declarative language for synchronous programming of real-time systems. In *Proc. of a conf. on Functional programming languages and computer architecture*, pages 257–277, 1987.

[67] M. Geilen and T. Basten. Requirements on the Execution of Kahn Process Networks. In *Proceedings of the 12th European Symposium on Programming (ESOP)*, 2003.

[68] E. Goldberg, M. Prasad, and R. Brayton. Using sat for combinational equivalence checking. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '01, pages 114–121, Piscataway, NJ, USA, 2001. IEEE Press.

[69] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Proceedings of 7th International Workshop on Hardware/Software Co-Design, CODES'99*, Rome, Italy, May 1999.

[70] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the Message Passing Interface Standard. *J. Parallel Computing*, 22(6):789–828, Set. 1996.

[71] P. L. Guernic, M. Borgue, T. Gautier, and C. Marie. Programming real time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1335, 1991.

[72] P. L. Guernic, J.-P. Talpin, and J.-C. L. Lann. Polychrony for system design. *J. of Circuits, Systems, and Computers - Sp. Is.: Application Specific Hardware Design*, 12(3):261–303, Dec. 2003.

[73] N. Halbwachs. Synchronous Programming of Reactive systems. *Kluwer Academic Publishers, Netherlands*, 1993.

[74] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, Sept. 1991.

[75] N. Halbwachs and F. Maraninchi. On the symbolic analysis of combinational loops in circuits and synchronous programs. In *Euromicro'95*, Como (Italy), September 1995.

[76] N. Halbwachs and P. Raymond. A tutorial of lustre, 2001.

[77] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[78] C. A. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, Oct. 1974.

[79] Intel Corporation. Intel Multi-core technology. http://www.intel.com/multi-core/.

[80] Intel Multicore Resource Center. The route to multithreading, the drive towards parallelism. http://www.developers.net/intelmcshowcase.

[81] P. Jackson and J. Pais. Computing prime implicants. In *CADE-10: Proc. of the tenth intl. conf. on automated deduction*, pages 543–557, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[82] B. A. Jose. New Redundant Binary Adder Cells for Multiplier Design. 2006.

[83] B. A. Jose, A. Gamatie, M. Kracht, and S. K. Shukla. Improved False Causal Loop Detection in Polychronous Specification of Embedded Software. http://fermat.ece.vt.edu/Publications/pubs/techrep/techrep1108.pdf, 2011. FERMAT Technical Report 2011-08.

[84] B. A. Jose, A. Gamatie, J. Ouy, and S. K. Shukla. SMT Based False Causal loop Detection during Code Synthesis from Polychronous Specifications. http://fermat.ece.vt.edu/Publications/pubs/techrep/techrep1104.pdf, July 2011. Proc. of 9th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE).

[85] B. A. Jose, H. D. Patel, S. K. Shukla, and J.-P. Talpin. Generating multi-threaded code from polychronous specifications. *Electron. Notes Theor. Comput. Sci.*, 238(1):57–69, 2009.

[86] B. A. Jose, J. Pribble, and S. K. Shukla. Faster embedded software synthesis using actor elimination techniques for multi-rate synchronous formalism. In *Proc. of 10th Conf. on Application of Concurrency to System Design (ACSD)*, pages 147–156, June 2010.

[87] B. A. Jose, J. Pribble, and S. K. Shukla. Technical Report on MRICDF models. https://filebox.vt.edu/users/bijoyaj/files/mricdfmodels.pdf, 2010. FERMAT Technical Report 2010-01.

[88] B. A. Jose, J. Pribble, L. Stewart, and S. K. Shukla. EmCodeSyn: A Visual Framework for Multi-Rate Data flow Specifications and Code Synthesis for Embedded Application. *12th IEEE Forum on specification and Design Languages (FDL'09)*, Sept. 2009.

[89] B. A. Jose and D. Radhakrishnan. Delay optimized redundant binary adders. In *Electronics, Circuits and Systems, 2006. ICECS '06. 13th IEEE International Conference on*, pages 514 –517, dec. 2006.

[90] B. A. Jose and D. Radhakrishnan. Fast redundant binary partial product generators for booth multiplication. In *Proceedings of the IEEE 50th Midwest Symposium on Circuits and Systems*, pages 297–300, 2007.

[91] B. A. Jose and D. Radhakrishnan. Redundant binary partial product generators for compact accumulation in booth multipliers. *Microelectronics Journal*, 40:1606–1612, November 2009.

[92] B. A. Jose, M. R. Sabharwal, and A. Agrawal. Power implications of high resolution timer tick settings. In *Energy Aware Computing (ICEAC), 2010 International Conference on*, pages 1–2, Dec. 2010.

[93] B. A. Jose and S. K. Shukla. An Alternative Polychronous Model and Synthesis Methodology for Model-Driven Embedded Software. *Proc. of IEEE Asia and South Pacific Design Automation Conf. (ASP-DAC 2010)*, 2010.

[94] B. A. Jose and S. K. Shukla. MRICDF : A polychronous Model for Embedded Software Synthesis. *Synthesis of embedded software - frameworks and methodologies for correctness by construction software design*, 2010.

[95] B. A. Jose and S. K. Shukla. New Techniques for Sequential Software Synthesis from a Polychronous Data Flow Formalism. http://www.fermat.ece.vt.edu/Publications/pubs/techrep/MRICDFJournal-techrep.pdf, 2011. FERMAT Technical Report 2011-07.

[96] B. A. Jose, S. K. Shukla, H. D. Patel, and J.-P. Talpin. Algorithm and Grammar for Multithreading SIGNAL. http://fermat.ece.vt.edu/Publications/pubs/ techrep/techrep0804.pdf.

[97] B. A. Jose, S. K. Shukla, H. D. Patel, and J.-P. Talpin. On the deterministic multi-threaded software synthesis from polychronous specifications. In *Formal Methods and Models for Co-Design, 2008. MEMOCODE 2008. 6th ACM/IEEE International Conference on*, pages 129–138, June 2008.

[98] B. A. Jose, S. K. Shukla, and J.-P. Talpin. MRICDF: A New Polychronous Model of Computation for Reactive Embedded Software. http://fermat.ece.vt.edu/Publications/pubs/ techrep/techrep0805.pdf.

[99] B. A. Jose, L. Stewart, J. Pribble, and S. K. Shukla. Technical report on EmCodeSyn models: STARMAC and Producer-Consumer examples. Technical Report 2009-02, FERMAT Lab, Virginia Tech, 2009.

[100] B. A. Jose, B. Xue, and S. K. Shukla. An analysis of the composition of synchronous systems. *Electron. Notes Theor. Comput. Sci.*, 245:69–84, 2009.

[101] B. A. Jose, B. Xue, S. K. Shukla, and J.-P. Talpin. Programming Models for Multi-Core Embedded Software. *Multi-Core embedded systems*, 2010.

[102] G. Kahn. The Semantics of a Simple Language for Parallel Programming. *Proceedings of Information Processing*, pages 471–475, 1974.

[103] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[104] E. A. Lee and T. M. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[105] O. Maffeis and P. L. Guernic. Distributed Implementation of Signal: Scheduling & Graph Clustering. *3rd International School and Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 547–566, September 1994.

[106] O. Maffeis and P. Le Guernic. *Distributed Implementation of SIGNAL: Scheduling & Graph Clustering. In Lec. Notes in Comp. Sc.*, volume 863. Springer-Verlag Publishers, 1994.

[107] S. Malik. Analysis of cyclic combinational circuits. In *Proc. of the IEEE/ACM Intl. Conf. on Computer-aided design*, pages 618–625, 1993.

[108] M. Maroti, A. Ledeczi, and P. Volgyesi. The generic modeling environment. In *The IEEE Workshop on Intelligent Signal Processing (WISP'01)*, 2001.

[109] A. Matusiewicz, N. Murray, and E. Rosenthal. Prime Implicate Tries . In *Proc. of 18th intl. conf. on automated reasoning with analytic tableaux and related methods, Lecture Notes in Computer Science*, volume 5607/2009, pages 250–264, Oslo, Norway, 2009.

[110] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr 1989.

[111] K. S. Namjoshi and R. P. Kurshan. Efficient analysis of cyclic definitions. In *Proceedings of the 11th International Conference on Computer Aided Verification*, CAV '99, pages 394–405. Springer-Verlag, 1999.

[112] M. Nanjundappa, H. D. Patel, B. A. Jose, and S. K. Shukla. Scgpsim: A fast systemc simulator on gpus. In *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pages 149 –154, Jan. 2010.

[113] D. Nowak. Synchronous structures. *Inf. Comput.*, 204(8):1295–1324, 2006.

[114] J. Ouy, J.-P. Talpin, L. Besnard, and P. Le Guernic. Separate compilation of polychronous specifications. In *Formal Methods for Globally Asynchronous Locally Synchronous Design (FMGALS'07)*. Elsevier Electronic Lecture Notes on Computer Science, 2007.

[115] T. M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California, Berkeley, 1995.

[116] C. Petri. *Kommunikation mit automaten*. PhD thesis, Technische Universitat Darmstadt, Germany., 1962.

[117] D. Potop-Butucaru. Optimizations for faster simulation of Esterel programs. *Ph.D. thesis, Ecole des Mines*, 2002.

[118] D. Potop-Butucaru and B. Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specifications. *Application of Concurrency to System Design, 2005. ACSD 2005. Fifth International Conference on*, pages 48–57, June 2005.

[119] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. *Form. Methods Syst. Des.*, 28(2):111–130, 2006.

[120] D. Potop-Butucaru, R. de Simone, and Y. Sorel. Necessary and sufficient conditions for deterministic desynchronization. In *EMSOFT '07: Proc. of the 7th ACM & IEEE Intl. Conf. on Embedded software*, pages 124–133, New York, NY, USA, 2007. ACM.

[121] D. Potop-Butucaru, S. Edwards, and G. Berry. Compiling esterel. *Springer*, pages 3–14, 2007.

[122] J. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In M. Dezani-Ciancaglini and U. Montanari, editors, *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer Berlin / Heidelberg, 1982.

[123] M. D. Riedel and J. Bruck. The synthesis of cyclic combinational circuits. In *In Design Automation Conference (DAC*, pages 163–168. ACM, 2003.

[124] K. Schneider. The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, December 2009.

[125] K. Schneider and T. Schuele. Averest: Specification, verification, and implementation of reactive systems. In J. Desel and Y. Watanabe, editors, *Application of Concurrency to System Design (ACSD)*, St. Malo, France, 2005. Participant's proceedings.

[126] E. Sentovich. Quick conservative causality analysis. In *System Synthesis, 1997. Proceedings., Tenth International Symposium on*, pages 2 –8, sep 1997.

[127] T. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *Proceedings of the 1996 European conference on Design and Test*, pages 328–, 1996.

[128] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *Proc.of the 37th ACM SIGPLAN-SIGACT symp. on Principles of programming languages*, POPL'10, pages 313–326, 2010.

[129] S. Suhaib. *Formal Methods for Intellectual Property Composition Across Synchronization Domains*. PhD thesis, Virginia Polytechnic Institute and State University, 2007.

[130] S. Suhaib, B. A. Jose, S. K. Shukla, and D. Mathaikutty. Formal transformation of a kpn specification to a gals implementation. In *IEEE Forum on Specification, Verification and Design Languages (FDL)*, pages 84 –89, sept. 2008.

[131] Synfora Inc. Esterel Studio EDA Tool. http://www.synfora.com/products/esterelStudio.html.

[132] J.-P. Talpin, J. Ouy, L. Besnard, and P. L. Guernic. Compositional design of isochronous systems. *Design Analysis and Test in Europe (DATE'08)*, pages 928–933, March 2008.

[133] O. Tardieu and R. de Simone. Curing schizophrenia by program rewriting in esterel. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pages 39 – 48, june 2004.

[134] O. Tardieu and R. D. Simone. Instantaneous termination in pure esterel. In *Proceedings of the 10th international conference on Static analysis*, SAS'03, pages 91–108. Springer-Verlag, 2003.

[135] G. Taubenfeld. Synchronization Algorithms and Concurrent Programming. *Pearson Education limited, England*, 2006.

[136] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A Language for Streaming Applications. *In Proceedings of the 11th international Conference on Compiler Construction, Lecture Notes In Computer Science, Springer-Verlag, London*, 2304:179–196, Apr. 2002.

[137] R. Traussnig and H. Giese. Model-based software development and automated code generation for safety-critical systems. http://www2.cs.uni-paderborn.de/cs/ag-schaefer/Lehre/Lehrveranstaltungen/Seminare/ATSEfSCS/slides /Automated%20Code%20Generation.pdf, 2004.

[138] U.S. Department of Transportation Federal Aviation Administration. Assessment of Software Development Tools for Safety-Critical, Real-Time Systems. http://www.tc.faa.gov/its/worldpac/techrpt/ar0636.pdf, 2007.

[139] L. H. Yoong, P. Roop, Z. Salcic, and F. Gruian. Compiling Esterel for Distributed Execution. *In Proceedings of Synchronous Languages, Applications, and Programming (SLAP)*, 2006.