Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 245 (2009) 69–84
www.elsevier.com/locate/entcs

An Analysis of the Composition of Synchronous Systems

Bijoy A. Jose [1] Bin Xue [2] Sandeep K. Shukla [3]

FERMAT Lab
Virginia Polytechnic Institute and State University
Blacksburg, VA, USA

Abstract

Safety-critical embedded applications are often distributed. For example, software in an automotive control or in avionics control are distributed over a large number of distributed processors which are connected over some domain specific buses. Correctness of such applications is of paramount importance due to their safety-critical nature. Synchronous programming models (e.g. Esterel, SIGNAL, Lustre) make synchrony assumption (zero time intra-module computation and zero time inter module communication) while modeling such applications so that the model is 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 polychronous programming model (exemplified by the SIGNAL language), a property called 'endo-isochrony' was proposed in early 1990s. Endochrony of individual modules assures safe sequential code generation from the module specification, and isochrony ensures safe communication between modules. In this paper, first we provide a more general sufficient condition for isochrony. Second, we generalize the definition of isochrony for weakly-endochronous modules. Further, we introduce the notion of directional isochrony which provides sufficient conditions for safe communication between modules in one direction but not in the other direction. The results in this paper not only simplifies the understanding of the conditions under which a polychronous specification can be implemented in GALS, but also sheds interesting lights on causality and isochrony. When the synchronous modules are reused as IPs, the conditions described here can be checked to see whether those modules can be composed asynchronously with the same behavior as their synchronous composition.

Keywords: Synchronous Programming Model, Polychrony, SIGNAL, endochrony, isochrony, Globally Asynchronous Locally Synchronous Systems, Unidirectional Isochrony

1 Introduction

Globally Asynchronous and Locally Synchronous (GALS) designs are gaining importance in the field of System-on-Chip (SoC) design due to two facts: first, due to rising clock speed, and increased number of components in today's designs, signals

[1] Email: bijoy@vt.edu
[2] Email: xbin114@vt.edu
[3] Email: shukla@vt.edu

doi:10.1016/j.entcs.2009.07.029

reaching one end of the chip from another end within a clock cycle is becoming difficult; second, the power consumed in the clock distribution network is too much compared to the power consumed in computation. Therefore, a push towards multiple clock domains, and signals crossing clock domains have given rise to interest in GALS design. Since it is widely accepted that synchronous design is much easier than asynchronous designs, it is important to develop methodologies and tools for GALS design.

However, there is another kind of GALS design which has been in existence much before the application of GALS in SoC designs have become critical. In the early 1980s, various synchronous programming languages such as Esterel, Lustre, SIGNAL etc., were developed to simplify the design of embedded control software using a simplifying abstraction about time taken to compute and communicate. Under the synchrony assumption, building models for the control related computation is easier. It is also easier to verify such models for their computational correctness. However, the synchrony assumption is quite strong in assuming that the computation takes zero time, and the communication between synchronous modules takes zero time. Similar to the case of hardware designs, the computational zero-time can be easily validated if the inter-arrival time between subsequent inputs (or subsequent sampling of inputs) is greater than the computation time. However, when the modules are distributed across an asychronous communication media, such as a CAN bus (in automotive applications), the zero-time communication, and synchronous communication for all the signals is not easily justifiable. This gave rise to a large amount of research in the 1990s. Various conditions on the synchronous models under which an entire system can have the same computational behavior, despite the asynchronous media were invented.

In [2,3] one such condition called 'endo-isochrony' was introduced, which if satisfied by a collection of synchronous modules, would guarantee that the distributed deployment of the modules across asynchronous media would be safe. By being 'safe' we mean that the dataflow behavior of the collection of modules when working under synchrony assumption, and when working in asynchronously communicating scenario, would be the same. This notion of equivalence between the GALS deployment and the original synchronous models has been called 'flow equivalence' [3]. In this paper, we assume that whenever we say a synchronous model and its GALS implementation are equivalent, we mean 'flow equivalent'.

In [2,3], there are two parts to this condition. First, the individual modules must be 'endochronous'. Informally, this means that module specifications should have enough information in themselves so that when compiled into sequential software components, they are capable of sampling their inputs exactly when required without missing any input changes. In endochrony, sequentiality has been a prime pre-condition, leading to a restrictive class of synchronous models which satisfy this condition. In later works [16], 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 implementation were allowed, leading to the notion of weak-endochrony. The class of weak-endochronous

modules strictly subsumes the class of endochronous modules. Weak-endochronous modules has enough information in their specifications so that their multi-threaded or concurrent implementations can sample all inputs exactly when required without missing any input changes. Therefore, weak endochrony is a weaker restriction on modules.

The idea of isochrony on the other hand is not about the modules themselves, but about communication between modules. In [2,3], this idea was formulated in two ways. The first way was about reconstructing the synchronous communication from asynchronous trace of their communication. A second formulation was a sufficient condition and can be explained as follows. If two modules are communicating via multiple signals, then the signals must be in phase. Suppose modules M_1 and M_2 are communicating via signals x, y, z, w such that M_1 computes values for signals x, y, and M_2 consumes those values, and z, w values are produced by M_2 while M_1 consumes those values. Then in order for the communication between M_1 and M_2 to be considered isochronous, it must be the case, that M_1 must always produce values on x, y at the same instants, and same is true for M_2 while producing z, w. If this happens, then M_1 would always wait for a value of y to arrive, if a new value of x arrives and vice versa. Therefore, M_2 can reconstruct the synchronized behavior of production of x, y without having to get any specialized handshake signals from M_1 and vice versa. This is however, restrictive and modules that were designed to be working in a zero-time computation and communication may not easily guarantee this condition. It also requires model checking to check that this holds. If this holds, then M_1 and M_2 could be easily put across asynchronous media with suitable flow control protocol between them.

In this paper, we generalize the sufficiency conditions in [2,3] to give a characterization of isochrony in terms of clock relations between signals. In another direction, we provide a sufficiency condition for weakly endochronous modules which requires a modification of the definition used for endochronous components. In [17] weak-endochronous module composition and the corresponding construction to make the entire composition isochronous was considered. However, our goal is to find ways to check for a collection of weakly endochronous modules, if they satisfy the modified isochrony condition such that their composition is safely implementable in a GALS setting.

So, in comparison with previous work, in the paper, we find conditions on clock relations either directly coded as clock constraints, or indirectly inferred from the dataflow equations, which would imply that the GALS implementation of the original synchronous model under analysis is correct. In this paper, we do not go into the algorithms for doing so, but rather stop at providing the sufficiency characterization of isochrony in terms of clock relations.

Before we proceed further, we must briefly discuss the programming model used in this work. As mentioned before, there are multiple variants of synchronous programming model. Esterel [4] is an imperative paradigm of programming that makes synchrony assumption. Lustre [9] on the otherhand describes dataflow equations to describe the computation inherent in an embedded application. The Lustre se-

mantics assumes that these equations are evaluated based on the global clock tick, and hence suitable for distributed applications that implements a global clock synchronization such as Time Triggered Architectures [12]. Our work is based on the SIGNAL [8] language and the Polychrony framework [7]. SIGNAL is a dataflow synchronous language closer in flavor to Lustre. However, instead of dataflow equations, SIGNAL specifies dataflow relations (and hence SIGNAL is more in the specification domain than programming language). Also, SIGNAL does not assume any global clock to pace the computation of the dataflows, but rather each signal in a SIGNAL model is sampled or computed based on its own rate (clock). These rates are often related by clock relations which are either inferred from the dataflow relations (usage) or by direct clock constraints written in the model. These rates are then analyzed using clock calculus [1], and the property of endochrony is basically determined by the clock calculus. Therefore, it is possible to check endochrony (albeit with high complexity) by solving Boolean relations encoding the clock relations. This paper aims to set the foundations for the same in analyzing isochrony.

The analysis and contributions in this paper are arranged as follows. The theoretical background for synchronous languages in general and SIGNAL in particular are given in Section 2. The operators in SIGNAL and the code generation strategy using rate relations are discussed. We also define the rate relations in SIGNAL and introduce our formalism to represent them in this section. Section 3 contains the assertions which govern the allowed composition requirements and examines them with respect to the properties of SIGNAL models. The communicating signals and their structures are defined which will be used throughout the paper to explain other contributions. The directional composition of components with examples which show correct and incorrect cases are given in Section 4. An important step in the transition from a dataflow based design to a GALS implementation is the code generation part. Section 5 will review the code generation proposals for distributed implementation and will comment on the related work. Section 6 will conclude the paper by detailing the contributions of this paper.

2 SIGNAL Modeling and Rate relations

Dataflow synchronous language such as SIGNAL expresses computation by capturing dataflow in terms of relations, and by capturing syntactic modularization by process constructs. So in SIGNAL, one can write process descriptions which has inputs and outputs and internal variables. The variables are called signals, because the variables in a SIGNAL process semantically denote infinite sequences of values that the variables take over time. $\hat{x}$ defines the rate (clock) at which a signal x will get updated or evaluated. The signals can have various data types such as integers, booleans, real numbers, events etc. The computation is expressed as concurrent activity on the various signals. There are four basic operator types (one can define many derived operators based on them): functions, sampler, merge and delay. All possible dataflow functions (for which computational library elements are available) can be used.

Usually a `Function` operator f is applied on a set of signals, say $x_1, ..., x_n$ to produce value on a signal y and is written as:

$$y := f(x_1, ..., x_n) \tag{1}$$

This means that a new value of y is computed by function f by sampling the values at signals x_1,x_2,...,x_n. It is the responsibility of the modeler to ensure that the values of each signal is present at its rate. In other words, the usage of signals by the modeler itself will indicate to the compiler that rates at which the values of x_1,x_2,...,x_n are updated are the same, and y gets updated at the same rate, as by synchrony hypothesis, f computes in zero time. The SIGNAL compiler assumes the environment will supply the inputs at each clock instant and will show errors in case the clock calculus reveals this cannot be guaranteed.

The `Sampler` operator is used as follows:

$$y := x \; when \; c \tag{2}$$

where c is a Boolean condition on some signals, and y and x must have the same data types. This statement means that whenever the condition c is computed, or sampled, and turns out to be 'true', if at that time, x also has a valid value, then y will immediately get that value. If however c is evaluated to false, then y will not get a new value. So the rate at which y is updated is less than the rate at which x is updated, or the rate at which c is evaluated to true.

The `Merge` operator is called 'default' and used as follows:

$$y := x \; default \; z \tag{3}$$

where x, y, and z must have the same data type, and whenever x gets sampled or evaluated, y gets the value of x. If x is not evaluated or updated but z does get evaluated or updated, then y gets that value. So it is a deterministic merge with priority to the signal x over z.

The `Delay` operator is denoted by $, and an example usage would be as follows:

$$y := x\$ \; init \; k \tag{4}$$

This means that y gets the previous value of x every time x gets updated. In the first instance, y is initialized with k.

A `Function` operator from 1 will equate the rate of the resultant and input signals. Similar is the case for `Delay` operator, which equates the rates of the input and output signals. The rate of y in Equation 2 can be represented as an intersection between the clock of x and the clock of $[c](c$ is present and true). Similarly, we can observe that a union of the clocks of x and z occurs while using a `Merge` operator in Equation 3. The clock relations between input and output signals used along with an operator in SIGNAL is summarized in Table 1.

So a system of dataflow relations represents a dataflow network. However, there is a notion of causality in a network which decides the order in which each set of

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	$y = x\ when\ c$	$\widehat{y} = \widehat{x} \cap [c]$
Merge	$y = x\ default\ z$	$\widehat{y} = \widehat{x} \cup \widehat{z}$
Delay	$y = x\ \$\ init\ k$	$\widehat{y} = \widehat{x}$

Table 1
SIGNAL Operators and their clock relations

signals have to be evaluated. Even though the statements in SIGNAL are in parallel composition, there exists dependency between some of the signals which necessitates this ordering. This causality is caused by boolean gaurds in Sampler operator, availability of inputs etc. which gives rise to rate relations. We are interested in the evaluation of these rate relations and how they influence the communication between components.

SIGNAL language has a syntactic construct called process which is its way of modularizing the dataflow specification. The processes are assumed concurrent modules. As a result, if process boundaries are deleted, one gets a large dataflow network where operators are connected with some other operators via signals. The signals at the boundaries of modules or processes also communicate in zero time as per synchrony assumption. However, when distributed in a GALS environment, usually separate modules are deployed at different sites. As a result, signals that cross the module boundaries call for special attention, because the zero time communication for these signals cannot be assumed any more. The concept of isochrony concerns particularly these signals.

Also it is worth noting that the data flow network does not have to be syntactically acyclic. This could be puzzling because if there is a cycle inside a zero time computation, it seems to be nonimplementable. However, one has to remember that not all operators are computing at the same time point as discussed earlier. So if a cycle is such that not all operators on the same cycle in the dataflow network compute at the same rate, one can actually implement the dataflow without any problem. These are called constructive cycles.

So the clocks of the signals computed by the operators must be related to check for this condition. When signals cross boundaries of modules, this is particularly problematic to check, because the modules may be coming from different modelers, and may already be compiled into code. Hence, the clock relations of all signals inside each module must be exported along with the compiled module when exporting a module as an IP. The clock relations between different modules should be analyzed using clock calculus to check for such cycles. In the rest of this paper, we assume that such cycles have been checked before trying to create GALS implementation of the model.

Let us now look at a sample SIGNAL program for a better understanding of the paradigm.

```
process Sampler = (? integer a; event b;  ! integer c;)
                  (| c := a when ^b |)
```

The SIGNAL program `Sampler` uses the *when* primitive to sample the input signal a with another input b. The sampled output is denoted by c. Here the rate of the three signals can be different, but the rate of c is determined by the rates of a and b. The rate (clock) of a signal a is denoted by $\hat{a}$. The rate relation of `Sampler` output c is as follows:

$$\hat{c} = \hat{a} \cap \hat{b} \tag{5}$$

We now formally define *clock relation* in the SIGNAL context, which will be used later in the paper to define communication requirements, isochrony etc.

Definition 2.1 *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 2.2 *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 in terms of two other signals (as in Sampler/Merge), 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 $x := y\ default\ z$, $r_{xyz} = (\widehat{x} = \widehat{y} \cup \widehat{z})$.*

In the Table 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. This new relation can be used to assert that there exists no direct or indirect relation between the clocks of the signals under consideration.

Definition 2.3 *The signals x and y are independent if and only if the clock relation $r_{xy} = (\widehat{x} \ \#\widehat{y})$.*

SIGNAL programs are synthesized into ANSI C code using the Polychrony compiler [7]. 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 prop-

erty 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 [11]. 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 [3] 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. This [2,3] 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 [2]. From [2],

$$(\Phi_1 \mid\mid \Phi_2)^a \;=\; \Phi_1^a \mid\mid_a \Phi_2^a \,^4 \tag{6}$$

This definition of isochrony was shown to hold when two STSs Φ_1, and Φ_2, and Φ_2 having a set of common variables have 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 a sufficient condition but 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

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

understanding of isochrony substantially compared to [2,3]. Second, it generalizes it to weakly endochronous modules also. Third, a directional notion of isochrony comes up naturally.

3 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 throughout the course of the paper 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.

3.1 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 [3]. 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 defintions which will check for isochrony in composing processes, we revisit some information that we know from existing literature in the form of assertions. From [3], 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 [3] 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 isochronoy. 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 3.1 *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 3.1, 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;
```

3.2 Composition of weakly endochronous processes

A process is said to be weakly endochronous if it satisfies the *diamond property* defined in [16]. 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 [16] 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 3.2 *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 there is none and 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 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 4.1 *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 correcness 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 4.2 *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 *For two composed processes P_i and P_j, $R'_{ij} \preceq R'_{ji}$ if and only if $\forall r^{ij}_{ab} \in R'_{ij}$ and its corresponding $r^{ji}_{ab} \in R'_{ji}$, there exists a partial order such that $r^{ij}_{ab} \leq r^{ji}_{ab}$.*

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.

5 Related work and Code generation opportunities

Isochrony and the compositionality of endo-isochronous systems were discussed in [2]. They also gave the necessary and sufficient conditions for the equivalence of synchronous and asynchronous composition. Related work based on this property has attempted to model asynchronous implementations of synchronous specifications [15]. 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. A code generating strategy by separate compilation of polychronous specifications and utilization of weak endochrony for composition has been proposed in [14]. Another strategy for the composition of isochronous systems by weakening the endochrony property is discussed in [17]. They perform formal analysis on loosely time-triggered architecture to show that isochrony can be maintained in a composition of non-endochronous components.

In other synchronous languages, different code generation strategies for incorporating distributed operation has been proposed. In Esterel, code has been divided into atomic tasks which is scheduled with parallelism by referring a linked list containing the dependencies [6]. More focus is given to the study of compositionality of processes which will later help in designing compilers with the goal of distributed implementation. In the SIGNAL language, one of the earliest work was regarding the clustering and schedulability of SIGNAL programs discussed in [13]. A prominent work in distributed execution of synchronous languages was done by converting them into a common Object Code (OC) and then parallelizing this code into the many computing sites [5]. In this work, no characterization of the modules or the inter-module communication were done to check for distributed deployment. It was assumed that a global clock is implemented distributively by virtue of clock synchronization algorithms. However, since global clock synchronization is expen-

sive, in the SIGNAL literature or in this work, it is not considered. Instead, one seeks characterizations under which such expensive protocols would not be needed for GALS execution. Multi-threaded code generation remains an important milestone in achieving a separated, but centralized implementation for synchronous programs. In the current Polychrony compiler, only endochronous processes can be implemented. A beta version of multi-threaded code generating compiler is being developed. Process based threading for weakly endochronous programs with separate compilation for composing processes have been implemented using the existing compiler for SIGNAL [10] and a synchronous data-flow based threading strategy has been proposed in [11].

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.

6 Conclusions

Embedded software programming using synchronous programming model has been found to be suitable for GALS architectures. The challenges faced in using asynchronous communication between synchronous IPs requires more research into the composition of IPs. The concept of 'isochrony' has evolved over the years into a property which provides a sufficient condition for the correct composition of synchronous components. The properties pertaining to synchronous domain like endochrony, weak endochrony and how it affects isochrony needs to be studied for composition of synchronous systems. In this paper, we have formalized the conditions for isochronous composition and have proposed a communication structure involving the parameters required for composition. The assertions proposed based on the communication structure will serve as a sufficient test for the composition of IPs. We have shown that weak endochrony relaxes the strict conditions for composition of IPs and correctness can be guaranteed for communication towards a weakly endochronous process. A new null clock relation used in this paper has enabled us to define directional isochrony, which provides us with a special set of rules for correctness of communication in a single direction. Finally the implications of these properties for code generation for distributed architecture is discussed. IP reuse and addition of IPs into a GALS system will benefit from research into the isochronous composition of synchronous IPs.

Acknowledgement

We wish to thank Jean-Pierre Talpin (INRIA) and Dumitru Potop-Butucaru (INRIA) for useful discussions on the content of this paper. This work was partially supported by NSF grant CCF-0702316 and SRC task 1818.

References

[1] Amagbegnon, T. P., L. Besnard and P. L. Guernic, *Implementation of the data-flow synchronous language signal*, in: *Proceedings of the ACM Symposium on Programming Languages Design and Implementation (PLDI'95)* (1995), pp. 163–173.

[2] Benveniste, A., B. Caillaud and P. L. Guernic, *From synchrony to asynchrony*, in: *Proc. of Concurrency Theory*, 1664 (1999), pp. 162–177.

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

[4] Boussinot, F. and R. D. Simone, *The ESTEREL language*, Proc. of the IEEE **79** (1991), pp. 1293–1304.

[5] Caspi, P., A. Girault and D. Pilaud, *Automatic distribution of reactive systems for asynchronous networks of processors*, IEEE Transactions on Software Engineering **25** (1999), pp. 416–427, research report INRIA 3491.

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

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

[8] Guernic, P. L. and T. Gautier, *Data-flow to von neumann: the signal approach*, Advanced Topics in Data-Flow Computing (1991), pp. 413–438.

[9] Halbwachs, N., P. Caspi, P. Raymond and D. Pilaud, *The Synchronous Data-Flow Programming Language LUSTRE*, Proc. of the IEEE **79** (1991), pp. 1305–1320.

[10] Jose, B. A., H. D. Patel, S. K. Shukla and J.-P. Talpin, *Generating Multi-Threaded code from Polychronous Specifications*, in: *Proc. of Synchronous Languages, Applications, and Programming (SLAP)*, Budapest, Hungary, 2008.

[11] Jose, B. A., S. K. Shukla, H. D. Patel and J.-P. Talpin, *On the Deterministic Multi-threaded Software Synthesis from Polychronous Specifications*, in: *Formal Models and Methods in Co-Design (MEMOCODE'08)*, Anaheim, California, 2008.

[12] Kopetz, H. and G. Bauer, *The time-triggered architecture*, Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software (2001).

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

[14] Ouy, J., J.-P. Talpin, L. Besnard and P. L. Guernic, *Separate compilation of polychronous specifications*, in: *FMGALS '07: Formal Methods for Globally Asynchronous Locally Synchronous Design* (2007).

[15] Potop-Butucaru, D. 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 (2005), pp. 48–57.

[16] Potop-Butucaru, D., B. Caillaud and A. Benveniste, *Concurrency in synchronous systems*, Formal Methods in System Design **28** (2006), pp. 111–130.

[17] Talpin, J.-P., J. Ouy, L. Besnard and P. L. Guernic, *Compositional design of isochronous systems*, in: *DATE '08: Proc. of ACM & IEEE Intl. Conf. on Design and Test in Europe* (2008), pp. 928–933.